



Intermediate Python for Engineers

COURSE NOTES

Copyright ©2010-2020: Python Charmers Pty Ltd, Australia, and Python Charmers Pte Ltd, Singapore

Edition: 2020.10



All rights to text, illustrations and source code are reserved. This work may not be copied, reproduced, or translated in whole or in part without written permission of Python Charmers. Use with any form of information storage and retrieval, electronic adaptation or whatever, computer software, or by similar or dissimilar methods now known or developed in the future is strictly forbidden without written permission of the copyright holder.

Contents

1	Outline	1
1.1	Course details	1
1.2	Course outline	2
1.3	Instructor bios	4
1.4	Other information	7
1.5	About Python Charmers	8
1.6	Contact	8
2	Classes	9
2.1	Methods	10
2.2	Constructors	11
2.3	<code>self</code>	12
2.4	Properties	14
3	Lazy sequences	23
3.1	Iterables	23
3.2	Generators	24
3.3	Generator expressions	25
3.4	Generator functions	27
3.5	Examples	29
3.6	Iterators	34
3.7	Summary	39
3.8	Exercises: streaming analytics	40
4	Intermediate Python language features	43

4.1	Decorators	43
4.2	Context managers	47
4.3	Easily creating context managers	49
4.4	Extended exercise: Beyond PEP8	52
5	Managing memory	55
5.1	Variable names are just labels	55
5.2	Mutability	56
5.3	Mutable and immutable types	57
5.4	<code>object id</code>	58
5.5	Pass-by-reference or pass-by-value?	61
5.6	Garbage collection	64
6	Introduction to PyCharm	67
6.1	Creating a new project in PyCharm	68
6.2	The interpreter	70
6.3	Editing code	72
6.4	Running code in PyCharm	82
6.5	Debugging in PyCharm	84
6.6	Adding additional tools to PyCharm	86
7	Coding style	91
7.1	Maintainability	92
7.2	PEP8	94
7.3	PEP257 and the NumPy docstring standard	99
8	Linting	103
8.1	<code>pycodestyle</code>	104
8.2	<code>pylint</code>	107
8.3	Warnings vs errors	109
8.4	Take Pylint errors seriously	111
8.5	Take warnings seriously too!	111
9	Writing elegant Python	113
9.1	Iterate through sequences, not indices	114
9.2	Use <code>zip()</code> to iterate through sequences in lockstep	114
9.3	Using the (extended) unpacking syntax	117

9.4 Argument expansion in function calls	120
9.5 Use namedtuples (if immutable)	120
10 Writing elegant Python – part 2	125
10.1 Data Classes (if mutable)	125
10.2 Keep exception blocks small	128
10.3 Define <code>__iter__</code> on classes as a generator function	131
10.4 Use <code>islice</code> to extract the first n items from a generator	134
10.5 Use <code>itertools</code> for getting combinations of collections	135
10.6 Memoize the results of long running functions	138
10.7 Implementing a persistent cache	140
11 Packages and modules	145
11.1 Packages	145
11.2 How to find packages	145
11.3 Installing further packages	146
11.4 Modules	148
11.5 Creating modules	152
11.6 Packages in more depth	153
11.7 Importing from packages	153
12 Developing in Python	155
12.1 Conda environments	155
12.2 Getting packages	156
12.3 Creating and installing packages with <code>setup.py</code>	157
12.4 Writing command-line interfaces	158
13 Logging	161
13.1 Examples	162
13.2 Loggers, handlers, formatters, levels	164
13.3 Other logging packages	165
14 Unit testing	167
14.1 Setup	168
14.2 <code>pytest</code>	170
14.3 Parametrizing tests	171
14.4 Coverage	172

14.5 Property-based testing	173
14.6 Test fixtures	175
14.7 Mocking the the results of a function	176
15 Intro to test-driven development	181
15.1 Why would we want to write tests first?	181
15.2 How to write the tests first: Red, Green, Refactor	184
15.3 Step 1: Red	184
15.4 Step 2: Green	188
15.5 Step 3: Refactor	190
15.6 Best practices: exercise with TDD and decorators	193
16 Files and paths	195
16.1 Difficulties representing paths as strings	195
16.2 Path operations with <code>pathlib</code>	197
16.3 Reading text files	199
16.4 Reading binary files	200
16.5 Compressed data (zip, gzip, bz2, ...)	202
16.6 Low-level file operations	205
16.7 Cryptographic hashing (optional topic)	207
17 Working with IP addresses	209
17.1 Validating IP Addresses	209
17.2 Finding your local IP address	210
17.3 Finding adaptor info from Python	211
17.4 Finding properties of interfaces	212
17.5 IP networks	213
17.6 Finding subnet address blocks	214
17.7 Exercise: IP addresses on a subnet	214
18 Controlling external tools	217
18.1 Intro to Python's <code>subprocess</code> module	217
18.2 Interacting with open processes	219
18.3 Chaining processes together with pipes	222
18.4 Monitoring a log file	223
18.5 <code>sh</code> module	224

19 Process and system monitoring	225
19.1 Overview of psutil	225
19.2 Examples	225
19.3 Exercise: system monitoring	226
20 Templating with Jinja2	227
20.1 Loops	229
20.2 Environments	231
20.3 Filters	232
20.4 Inheritance	239
21 Python concurrency overview	245
21.1 The “minimum schedulable unit”	246
21.2 Asyncio	246
21.3 Python threads	247
21.4 GIL released threads	248
21.5 Multiprocessing	249
21.6 Distributed tasks	250
22 Threads	251
22.1 What is a thread	252
22.2 Creating a thread	252
22.3 Finding the memory usage	253
22.4 Simple thread example: slow IO	254
22.5 Running tasks on a thread	255
22.6 Which thread are we running on?	256
22.7 Locks	257
22.8 Thread pools	259
22.9 The global interpreter lock (GIL)	259
22.10 What to do instead?	265
23 Concurrency with concurrent.futures	269
23.1 Backported version	269
23.2 Scheduling a task to run concurrently	270
23.3 The Future class	271
23.4 A motivating example: a group of slow tasks	272
23.5 API calls: The slow synchronous way	273

23.6 Managing the flow of control: callbacks	275
23.7 Threadpool context manager	277
23.8 Map	278
23.9 Exercise: weather exercise with a thread pool	279
24 Parallel processing with Dask	281
24.1 Architecture	282
24.2 <code>dask.array</code>	282
24.3 <code>dask.dataframe</code>	288
24.4 <code>dask.bag</code>	295
24.5 <code>dask.delayed</code>	298
24.6 Schedulers	302
24.7 Distributed processing with Dask	303
24.8 Under the hood: task graphs	305
25 Web APIs	307
25.1 HTTP in a nutshell	307
25.2 URLs and Parameters	311
25.3 POST	320
25.4 Using the Telstra messaging APIs	322
26 Creating web apps with Flask	325
26.1 Getting started	325
26.2 Templates	328
26.3 Sessions	329
27 Best practices in Flask	335
27.1 Handling URL parameter types	335
27.2 Logging in Flask	337
27.3 Error handlers	338
27.4 Reversing URLs	339
28 Network automation	343
28.1 SSH automation with Fabric	343
28.2 Fabric Connection	344
28.3 Configuring fabric hosts	350
28.4 <code>fab</code> Command Line Interface	351

28.5 More libraries for network automation	353
29 Regular expressions	355
29.1 Regular expressions and the <code>re</code> module	355
29.2 Extract lines with specific log messages	359
29.3 Parsing dates and times	365
29.4 Other resources	367
30 Parsing semi-structured data	369
30.1 Intro to TextFSM	369
30.2 TextFSM templates	371
30.3 Example: a simple template	373
30.4 Exercise: parsing semi-structured text	376
30.5 Storing output in a Pandas DataFrame	377
30.6 Example matching multiple lines of text	378
30.7 Practical examples of TextFSM templates	381
30.8 Example: analyzing and joining results	382
31 Further resources	387
31.1 Python Charmers materials	387
31.2 Online Python resources	388
31.3 More exercises	391
31.4 Questions?	391
31.5 About Python Charmers	392

Chapter 1

Outline

1.1 Course details

Audience

A mixed group of software developers, engineers, devops and systems engineers.

Prerequisite

A working knowledge of *Python*.

Overview

This course will help you to understand intermediate-level Python language features, best practices for basic automation and building applications, and library tooling for system and network automation.

Skills

Classes, decorators, context managers, logging, debugging, functional tools, iterators and generators for handling large datasets; knowing the gotchas of memory use and performance of Python and Pandas code, best practices

for building maintainable codebases in teams; system automation; network automation.

Duration

6 sessions x ½ day (mornings)

Format

Each topic is a mixture of hands-on exercises and expert instruction.

1.2 Course outline

Session 1: Intermediate language features

Session 1 will teach you about important features for maximizing your productivity with Python including classes and iterators:

- Files, bytes, strings, and encodings
- Classes and objects
 - Named tuples. When (not) to use classes
 - Initialization, data classes, class methods, inheritance
 - Properties, special methods
- Iterators and generators, with applications to large datasets; *toolz*

Session 2: Further language features

Session 2 will teach you further Python language features and explain important distinctions and “gotchas” that help you with developing robust, efficient systems in Python:

- When & how to use decorators
- Context managers:
 - Managing state; streamlining exception handling
- Worked example: elegant code beyond PEP8 with context managers
- Managing memory
 - Mutability vs immutability

- Memory profiling; garbage collection
- Worked example: writing a disk-backed caching decorator

Session 3: Best practices

Session 3 teaches you some “best practices” for developing maintainable, robust systems in teams:

- *PyCharm* IDE; debugging tools
- Logging; debugging practices
- Python style and docstring conventions; linting: *pylint*, *Black*
- Type hinting
- Comprehensions and readability
- Creating modules and reusable packages; structuring projects

Session 4: Testing

Session 4 teaches you how to create automated tests for *Python* code:

- Why testing? Overview of test-driven development (TDD)
- Introduction to *pytest*
- Fixtures
- Coverage reports
- Testing best practices
- Introduction to mocking
- (Time permitting): Introduction to property-based testing

Session 5: System automation

Session 5 gives you a tour of the amazing standard library and important 3rd-party tools for automating various systems-level tasks with Python:

- Handling files & paths; dates, times, and IP addresses
- Compression; cryptographic hashing of file contents
- Automating local commands via *subprocess*
- Process and system monitoring with *psutil*
- Creating scripts with command-line arguments
- Automatically creating config files via templating with *Jinja2*

- Parsing various config files: JSON, YAML,INI, XML (topics on request)
- Concurrency and parallelism: opportunities and gotchas; *dask*

Session 6: Network automation

Session 6 shows you how to use and create web APIs and automate various network-related tasks:

- Accessing REST web APIs in depth (sessions, OAuth, ...)
- Creating REST web APIs with *Flask*
- Network automation via SSH with *Fabric*
- Parsing log files with regular expressions and *TextFSM*
- Automating network device configuration with *Netmiko*

Supplemental materials

We will supply you with printed course notes, cheat sheets, and a USB stick containing kitchen-sink Python installers for multiple platforms, solutions to the programming exercises, several written tutorials, and reference documentation on Python and the third-party packages covered in the course.

1.3 Instructor bios

Your trainers for the course will be selected from:

Dr Edward Schofield

Ed is the founder of Python Charmers in Australia and Singapore. Ed has trained over 2500 people from dozens of organizations in data science using Python, including Atlassian, Barclays, Cisco, CSIRO, Dolby, Harvard University, IMC, Singtel Optus, Oracle, Shell, Telstra, Toyota, Verizon, and Westpac. He is well-known in the Python community as a former release manager of *SciPy* and the author of the widely used *future* package. He organizes the Python user group in Melbourne and is sought-after as a speaker at conferences in Python and data science around the world.

Ed holds a PhD in machine learning from Imperial College London. He also holds BA and MA (Hons) degrees in mathematics and computer science from Trinity College, University of Cambridge. He has 20+ years of experience in programming, teaching, and public speaking.

Henry Walshaw

Henry has almost 15 years of experience in Python application development and has trained hundreds of people in how to use Python from organisations including AGL, the Bureau of Meteorology, ESRI, the NSW Department of Finance, National Australia Bank, and Telstra.

Henry's core technical expertise relates to the development and analysis of large scale spatial datasets (primarily using Python), and communicating this understanding to both subject matter experts and the general public.

Before joining Python Charmers, Henry worked in both government and industry — at Geoscience Australia, the Victorian Department of Sustainability and Environment, and the Environmental Protection Agency (EPA); as a consultant with Sinclair Knight Merz (SKM), a manager at we-do-IT, and as CTO of a startup. He holds a Bachelors in Computational Science.

Dr Robert Layton

Robert is the author of the book “Data Mining in Python”, published by O'Reilly. He provides analysis, consultancy, research and development work to businesses primarily using Python. Robert has worked with government, financial and security sectors, in both a consultancy and academic role. He is also a Research Fellow at the Internet Commerce Security Laboratory, Federation University Australia.

Robert is a regular contributor to the Python-based scikit-learn open source project for machine learning and writes regularly on data mining for a number of outlets. He has presented regularly at a number of international conferences in Python, data analysis, and its applications. He is also the author of the website learningtensorflow.com.

Dr Ned Letcher

Ned is a data scientist and software engineer who has helped a range of organisations in projects involving machine learning, natural language processing, information retrieval, and data visualisation. Ned has been using Python for data analysis, visualisation, machine learning, and web development for over 10 years. Ned is a contributor to the Plotly *Dash* library and an active member of the *Dash* community.

Ned has a PhD in computational linguistics from the Natural Language Processing group at the University of Melbourne. He also has a Bachelor of Arts (philosophy and linguistics) and a Bachelor of Science with Honours (computer science). Ned regularly presents at local meetups and organises the Melbourne Data Visualisation Meetup.

Dr Juan Nunez-Iglesias

Juan Nunez-Iglesias is co-author of the book *Elegant SciPy*, published by O'Reilly Media. Juan is a core developer of the *scikit-image* Python library, and has contributed to many others in the scientific Python ecosystem, including *SciPy*, *NetworkX*, and *Matplotlib*. He has taught and presented at the SciPy conference in Austin, EuroSciPy, PyCon Australia, the Advanced Scientific Programming in Python summer school, and Software Carpentry workshops.

Juan is a research fellow at Monash University, with interests in neuroscience and biological image analysis. He also has a particular interest in renewable energy and the environment.

Juan has Bachelor's degree in Biomedical Science from the University of Melbourne and both an MSc in Statistics and PhD in Computational Biology and Bioinformatics from the University of Southern California.

Errol Lloyd

Errol's background is in computational neuroscience. He has been using Python for modelling neurological systems, digital signal processing, data analysis, and empirical research for 7 years.

Prior to joining Python Charmers in 2020, Errol trained fellow researchers

from both the sciences and the humanities in a variety of software solutions to research problems, including data and natural language analysis in python, data visualisation, interactive dashboards with front-end javascript, and version control and collaboration with git and GitHub.

Errol is an advocate for open source software and reproducible research in science, and is passionate about empowering others to use code in enhancing their productivity. He is currently completing doctoral studies on visual processing in the brain at the University of Melbourne.

1.4 Other information

Exercises

There will be practical programming exercises throughout the course. These will be challenging and fun, and the solutions will be discussed after each exercise and provided as source code on the USB sticks. During the exercises, the trainer will offer individual help and suggestions.

Timing

The course will run from 8:00 to roughly 12:00 each day.

Personal help

Your trainer(s) will be available after the course each day for you to ask any one-on-one questions you like - whether about the course material and exercises or about specific problems you face in your work and how to use Python to solve them. We encourage you to have your own data sets ready to use if this is relevant.

Certificate of completion

We will provide you a certificate if you complete the course and successfully answer the majority of the exercise questions.

1.5 About Python Charmers

Python Charmers is a leading global provider of Python training, based in Australia and Singapore. Python Charmers specializes in teaching programming for data scientists, scientists, engineers, computer scientists, and quants in the Python language. Python Charmers' delighted training clients include Atlassian, the Australian Federal Police, Barclays, Bureau of Meteorology, Cisco, CSIRO, Dolby, EDF, Geoscience Australia, Primary Health Care, Shell, Singtel Optus, Telstra, Toyota, Verizon, Westpac, and Woolworths.

1.6 Contact

Phone +61 1300 963 160
Email info@pythoncharmers.com
Web <https://pythoncharmers.com>

Chapter 2

Classes

Classes in Python are custom data types. They can contain both data and functions. (Functions attached to a class are often called “methods”.)

Not every concept maps neatly onto one of Python’s built-in data types (`str`, `list`, `dict`, `set`, etc.). Classes allows us to create our own data types that represent the problems we are working on.

Here is a very simple class in Python:

```
class BankAccount:  
    pass
```

288

`pass` is the do-nothing statement. (It’s just needed here to preserve the block structure Python expects.)

Objects are instances of classes. We can create a new `BankAccount` object as follows:

```
checking = BankAccount()
```

291

We can define many instances of the same general concept. They share any functions defined by the class but have their own data (“attributes”). For instance, here is a second bank account object with its own `balance`:

```
bills_acct = BankAccount()  
bills_acct.name = 'Bill Gates'  
bills_acct.balance = 85.6e9
```

294

We can now define a function that takes an instance of `BankAccount` as the first argument, like this:

```
def withdraw(account, amount):  
    account.balance -= amount
```

295

This function can now operate on `BankAccount` objects:

```
withdraw(bills_acct, 100)  
  
bills_acct.balance
```

1

```
85600001000.0
```

This should be straightforward to understand. However, notice that `withdraw` doesn't make much sense as a free-standing function; it only operates on objects that have a `balance` attribute – like `BankAccount` instances.

2.1 Methods

For this reason, it makes more sense to define `withdraw` within the namespace of the `BankAccount` class like this:

```
class BankAccount:  
    def withdraw(account, amount):  
        account.balance -= amount  
  
bills_account = BankAccount()  
bills_account.name = 'Bill Gates'  
bills_account.balance = 85.6e9
```

298

In object-oriented programming, we refer to the `withdraw` function as a *method*. Methods are just functions attached to the class definition:

```
BankAccount.withdraw(bills_account, 1000)  
bills_account.balance
```

23

85599999000.0

However, the usual way to call “methods” (functions defined within classes) is like this:

```
bills_account.withdraw(1000)  
bills_account.balance
```

24

85599998000.0

This usage is more convenient. Python unravels this internally and calls `BankAccount.withdraw(savings, 1000)` as we did explicitly above.

The idea of object-oriented programming is to use methods to delegate responsibilities to the objects themselves as a way of managing complexity of codebases.

2.2 Constructors

So far, so good. However, above we needed to remember to define a balance manually after creating each new `BankAccount` object. We can automate this and take out the chance of error using an `__init__` method:

```
class BankAccount:  
    def __init__(account, balance=0):  
        account.balance = balance  
  
    def withdraw(account, amount):  
        account.balance -= amount
```

301

```
savings = BankAccount()
```

302

```
savings.balance
```

303

0

2.3 self

By convention, the first argument of methods is called `self`. (This is standard practice and strongly recommended by the Python Style Guide.) So we write the above like this:

```
class BankAccount:  
    def __init__(self, balance=0):  
        account.balance = balance  
  
    def withdraw(self, amount):  
        account.balance -= amount
```

301

Suppose we define another `BankAccount` object as follows:

```
stock_acct = BankAccount(50000)  
stock_acct.withdraw(55000)  
stock_acct.balance
```

4

-5000

Notice that these two `BankAccount` objects have their own data (`balance`). The `self` object allows the methods (which are shared) to access a particular instance's data.

Example 2: Circle class

Here is another simple example: a `Circle` class.

```
import math

class Circle:
    def __init__(self, radius, colour):
        self._colour = colour
        self._radius = radius

    def area(self):
        return math.pi * self._radius**2

    def perimeter(self):
        return 2 * math.pi * self._radius

a = Circle(7, 'red')
b = Circle(21, 'blue')
print(a.area(), b.area(), sep='\n')
```

153.93804002589985

1385.4423602330987

Above we see the use of an underscore prefix in the attribute names: `self._colour` etc. According to the Python Style Guide (PEP8), this is a weak indicator that the attribute is for “internal use”; there is a better way to use this class’s API than by reading or writing this attribute directly.

Exercise: classes

1. Define a class `Card` for playing cards. The initialization takes a rank and a suit.
2. Define a method `is_royal()` that returns `True` or `False`.
3. Add a `bj_value()` method, which returns the card’s value in Black-jack (counting royal cards as 10 and Ace as 1).

Here is some code you can use, that works with an appropriate `Card` class:

```
card1 = Card("Queen", "Hearts")

print(card1.is_royal()) # Should print "True"
```

```
print(card1.bj_value()) # Should print 10
print(type(card1.bj_value())) # Should print: int

card2 = Card("3", "Clubs")

print(card2.is_royal()) # Should print: False

print(card2.bj_value()) # Should print 3
print(type(card2.bj_value())) # Should print: int
```

Extended exercises: simulation

4. Use `random.sample()` to draw 1 million random “hands” of 3 cards from a standard deck of 52 cards.
5. What proportion of blackjack hands with 3 cards go “bust” (sum of values > 21)?
6. Rewrite the answers to questions (4) and (5) to use generators.

Check: The resulting percentage is approximately 37%.

See these files for solutions:

- `solutions/blackjack_simulation_1.py`
- `solutions/blackjack_simulation_2.py`
- `solutions/blackjack_simulation_3.py`

2.4 Properties

A common pattern in some languages like Java to expose an attribute of a object is to define *get* and *set* access methods for the attribute.

For example, you could define a *get* method for the `radius` attribute of a `circle` object in the following way.

```
class Circle(object):
    def __init__(self):
        self.__radius = 1.0

    def get_radius(self):
        print("Method: get radius")
        return self.__radius

c = Circle()
c.get_radius()
```

Method: get radius

1.0

With the property feature, you can implement the link between an attribute and a method, and the method can define a series of actions on the attribute.

All you need to do is to decorate a method with the built-in `@property`. The method name should be the same as the name of the property.

Using the property feature, the `get` method of a circle can be defined in the following way.

```
class Circle(object):
    def __init__(self):
        self.__radius = 1.0

    @property
    def radius(self):
        print("Method: get radius")
        return self.__radius

c = Circle()
c.radius
```

Method: get radius

1.0

Note that you use noun-based naming convention with the syntax, `object.attribute`, but still call a method associated with the property.

You have just defined a `get` method for `radius` property, but you haven't defined a `set` method for the attribute yet. So, when you try to set a value to the property, you get the following error message:

```
c.radius = 10.0
```

16

In fact, omitting a `set` method is one way to define a read-only property of an object.

To designate a new method as the `set` method for a property, you can decorate the method with `@attribute.setter`. For example:

```
class Circle(object):
    def __init__(self):
        self.__radius = 1.0

    @property
    def radius(self):
        print("Method: get radius")
        return self.__radius

    @radius.setter
    def radius(self, value):
        print("Method: set radius")
        self.__radius = value

c = Circle()
c.radius = 10.0
c.radius
```

18

```
Method: set radius
Method: get radius
```

```
10.0
```

Now, we can make the `Circle` class more solid in handling errors. We can

add additional code in the *set* method to ensure that the given `radius` is not negative.

```
class Circle(object):
    def __init__(self):
        self.__radius = 1.0

    @property
    def radius(self):
        print("Method: get radius")
        return self.__radius

    @radius.setter
    def radius(self, value):
        print("Method: set radius")
        if value < 0.0: raise ValueError("Negative radius
                                         given, %f."%value)
        self.__radius = value

c = Circle()
c.radius = 10.0
c.radius = -10.0
```

Method: set radius

Method: set radius

Ok, now let's add more intelligence to the `Circle` class. Here we introduce one more property, `area`, and make sure that both the `radius` and `area` of a circle have correct values.

```
import math

class Circle(object):
    def __init__(self):
        self.__radius = 1.0

    @property
    def radius(self):
```

```
print("Method: get radius")
return self.__radius

@radius.setter
def radius(self, value):
    print("Method: set radius")
    if value < 0.0: raise ValueError("Negative radius
        given, %f."%value)
    self.__radius = value

@property
def area(self):
    print("Method: get area")
    return self.__radius * self.__radius * math.pi

@area.setter
def area(self, value):
    print("Method: set area")
    if value < 0.0: raise ValueError("Negative area
        given, %f."%value)
    self.__radius = math.sqrt(value / math.pi)
```

```
c = Circle()
c.radius = 10.0
c.area
```

21

```
Method: set radius
Method: get area
314.1592653589793
```

```
c.area = 100.0
c.radius
```

22

```
Method: set area
Method: get radius
```

5.641895835477563

Note that a `Circle` object does not store its area as an attribute variable. Instead, it calculates a radius corresponding to the given area and store the radius internally. In this way, you can remove any risk to store inconsistent radius and area values in a `Circle` object.

You can also define a `delete` method for a property by decorating the method with `@attribute.deleter`.

```
import math

class Circle(object):
    def __init__(self):
        self.__radius = 1.0

    @property
    def radius(self):
        print("Method: get radius")
        return self.__radius

    @radius.setter
    def radius(self, value):
        print("Method: set radius")
        if value < 0.0: raise ValueError("Negative radius
            given, %f."%value)
        self.__radius = value

    @radius.deleter
    def radius(self):
        print("Method: del radius")
        del self.__radius

    @property
    def area(self):
        print("Method: get area")
        return self.__radius * self.__radius * math.pi

    @area.setter
```

```
def area(self, value):
    print("Method: set area")
    if value < 0.0: raise ValueError("Negative area
        given, %f."%value)
    self.__radius = math.sqrt(value / math.pi)
```

```
c = Circle()
c.radius = 10.0
print(c.radius)
```

25

```
Method: set radius
Method: get radius
10.0
```

```
del c.radius
print(c.radius)
```

26

```
Method: del radius
Method: get radius
```

Since you have deleted the `__radius` attribute by calling the deleter method, you cannot retrieve the radius value any more.

Example 2: BankAccount .age property

Here we revisit the `BankAccount` example from the introductory section on classes and show how an account holder's age can be computed (dynamically) despite accessing it without an explicit method call:

```
import datetime as dt
```

308

```
class BankAccount:
    def __init__(self, date_of_birth, balance=0):
        """
```

311

```
Pass date of birth as a datetime.date object
"""
self.balance = balance
self.date_of_birth = date_of_birth

def withdraw(self, amount):
    self.balance -= amount

@property
def age(self):
    """
    Return approximate age in years
    """
    return (dt.date.today() - self.date_of_birth).days /
        365
```

```
account = BankAccount(dt.date(1960, 1, 1))
```

316

```
account.age
```

317

```
59.654794520547945
```

Example: BankAccount name property

Here is another example of a “setter” used for validation. Here we disallow the name of a bank account from containing the Unicode snowman character:

```
class BankAccount:
    ... # __init__ and other methods as above

    @property
    def name(self):
        return self._name

    @name.setter
```

```
def name(self, newname):
    if '\u00a0' in newname:
        raise ValueError("name can't have \u00a0!")
    self._name = newname
```

Try it out like this:

```
savings = BankAccount(dt.date(1980, 1, 1))
savings.name = 'Homer'
savings.name = '\u00a0 Smith'
```

321

Summary: property

In brief, the property feature of Python allows you to make a series of actions whenever users try to access an attribute using the normal norm-based `object.attribute` syntax.

Chapter 3

Lazy sequences

3.1 Iterables

An **iterable** data type is one that can have its contents looped over in a `for` loop. Iterating over it yields each item one at a time.

For example, objects of these data types are iterable: `str`, `list`, `dict`, `range`. These objects are not iterable: `bool`, `int`.

What is the difference? `range` objects, strings etc. have an `__iter__` method:

```
myrange = range(5, 10)
'__iter__' in dir(myrange)
```

1

True

```
myint = 5
 '__iter__' in dir(myint)
```

1

False

Python calls an iterable object's `__iter__` method implicitly at the start of a `for` loop and in other iterable contexts like `in` (e.g. `'h' in mystring`).

Now consider this example:

```
file = open('/Data/eight_schools.csv')  
  
print(list(file))  
print(list(file))
```

1

```
[ 'School,Estimated Treatment Effect,Standard Error of  
    ↵ Treatment Effect  
' , 'A,28,15  
' , 'B,8,10  
' , 'C,-3,16  
' , 'D,7,11  
' , 'E,-1,9  
' , 'F,1,11  
' , 'G,18,10  
' , 'H,12,18'  
[] ]
```

The second `list()` call on the iterable file object produces an empty list.

Key point: Most iterators can only be wound in the forward direction. After looping through the generated values, no further iteration is possible. The iterator is exhausted.

3.2 Generators

When processing large datasets, it is a disadvantage to create a concrete sequence in memory. For example, if you are processing an input file that is hundreds or thousands of gigabytes, Python may try to consume too much memory to create a list, set, or dictionary.

For example: the following code creates a concrete list of all files on the filesystem – potentially millions or billions. It takes a long time (several minutes or even hours to traverse the whole filesystem) and uses lots of memory (to store a list of millions or billions of file paths).

```
import glob
```

```
allfiles = glob.glob('/**', recursive=True)      # slow; may fail
```

A variant of the `glob` function called `iglob` returns a **generator** instead of a concrete list:

```
allfiles = glob.iglob('/**', recursive=True)
```

The result (`allfiles`) is returned instantly. It is a generator:

```
type(allfiles)  
generator
```

A generator is a lazy sequence. Its elements are only generated when needed – usually when consumed in an iterable context like a `for` loop.

Here we show the first 5 files on the filesystem:

```
# Print files 0 to 4:  
for i, file in enumerate(allfiles):  
    print(file)  
    if i >= 5:  
        break
```

To show the next 5 files, we resume iterating over the generator as before:

```
# Print files 5 to 9:  
for i, file in enumerate(allfiles):  
    print(file)  
    if i >= 5:  
        break
```

Notice that the `allfiles` generator cannot be restarted. Past values cannot be retrieved. They are never stored in memory; instead they are yielded just-in-time and forgotten.

3.3 Generator expressions

Recall comprehensions like this:

```
squares = [k**2 for k in range(10)]
```

This creates a concrete list in memory.

Python offers simple syntax for creating a generator:

```
squares = (k**2 for k in range(10))
```

140

This comprehension syntax with parentheses does not create a tuple, as you might expect. It is a **generator expression**. The result is a generator:

```
type(squares)
```

141

```
generator
```

Passing the generator into the `list` constructor is equivalent to a list comprehension:

```
# These are equivalent:  
list(squares) == [k**2 for k in range(10)]
```

142

```
True
```

but the generator expression is the more general construct; it can also be used in other contexts that need not have all the data generated in memory at once.

In general, consider again this common pattern for transforming some data:

```
# Pseudo-code: this may exhaust memory:  
outputs = [process(line) for line in huge_file]
```

Provided that we only need to consume `outputs` once (not repeatedly), we can gain large memory benefits (and small performance benefits) by simply replacing the square brackets with parentheses:

```
# Pseudo-code:  
outputs = (process(line) for line in huge_file)
```

Now the function `process` will only be called on each line only when we iterate through `outputs`.

Example: iterating over all files

In this example, we lazily iterate over all files on a filesystem (potentially billions of files), retrieve the file size of each, and do something with each filesize (e.g. sum them up) without running out of memory:

```
import glob

# This returns a "generator" of all files in all subfolders, re-
paths = glob.iglob('/**', recursive=True)

# This "generator expression" also creates a generator, of thei-
filesizes = (getsize(path) for path in paths if isfile(path))
```

This code runs almost instantly and uses almost no memory. Python only walks your filesystem when you do something that iterates through the generated contents, like this:

```
sum(filesizes)
```

Exercise: a generator expression for leap years

1. Use a generator expression to create a generator that yields all leap years from 2000 to 2030.

Hint: use the `isleap()` function from the `calendar` module.

2. Try looping through all leap years and printing them.
3. Then try casting your generator to a list. Notice that the resulting list is empty. Why?

```
# See solutions/genexp.py
```

1

3.4 Generator functions

Although generator expressions are compact and convenient, they are not the most flexible way to define a generator. More powerful generators can be created using **generator functions**.

Here is a simple example:

```
def weekdays():
    DAYS = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
            ↵ 'Friday']
    while True:
        for day in DAYS:
            yield day
```

17

This looks just like a regular function except that it uses the `yield` keyword. `weekdays` is a “generator function”. It is a function that, when called, returns a generator.

```
type(weekdays)
```

18

function

```
w = weekdays()
type(w)
```

19

generator

What does this generator object generate?

```
next(w)
```

25

```
'Monday'
```

```
next(w)
```

32

```
'Tuesday'
```

`w` generates an infinite sequence of weekdays (because of the `while True` loop in `weekdays()`), so we cannot do this:

```
# infinitelist = list(w)
```

33

We can, however, loop over its contents and do useful things with each element.

Exercise: a generator function for leap years

Write a generator function `leap_year_gen(start, end)` that returns a generator for all leap years from `start` to `end`.

If you did the extended exercise above implementing a custom `LeapYearIterator` class, notice how much simpler this version is.

```
# See solutions/gen_ex1.py
```

1

3.5 Examples

Example 1: days of the year

```
from itertools import islice
```

34

```
# Assuming 1 January is Monday ...
last_day_of_year = islice(weekdays(), 364, 365)
```

35

```
list(last_day_of_year)
```

36

```
['Friday']
```

Generators (and other iterators) are often used paired up in flexible arrangements with other iterators. (The `itertools` module is particularly helpful with this.)

37

```
month_lengths = [('January', 31),
                  ('February', 28),    # assume not a leap
                  ↵   year
                  ('March', 31),
                  ('April', 30),
                  ('June', 31),
                  ('July', 30),
                  ('August', 31),
                  ('September', 30),
                  ('October', 31),
                  ('November', 30),
                  ('December', 31)]
```

38

```
# Another generator function:
def days_of_year():
    for (month, length) in month_lengths:
        for day in range(1, length+1):
            yield day, month
```

39

```
# Print the first day of each month:
for (weekday, (day, month)) in zip(weekdays(),
                                   ↵ days_of_year()):
    if day == 1:
        print(weekday, day, month)
```

```
Monday 1 January
Tuesday 1 February
Friday 1 March
Monday 1 April
Monday 1 June
Tuesday 1 July
Tuesday 1 August
Wednesday 1 September
Wednesday 1 October
Thursday 1 November
```

Thursday 1 December

Example 2: Walking your filesystem recursively

One benefit of iterators (and generators) is that they can be used “lazily”, with memory and speed advantages over creating big lists in memory.

Example: a common generator function is `os.walk` in the standard library:

```
# Find all folders containing jpg images, recursively:  
import os  
for (basedir, subdirs, files) in os.walk('/data'):  
    if any((f.endswith('.png')) for f in files):  
        print(basedir)
```

40

```
/data  
/data/images
```

Traversing all directories may take hours on a huge filesystem. But because `os.walk` returns a generator, it produces results lazily: there is no need to traverse all subfolders before the first result is returned.

Note: this is similar to streams in Unix. Quiz: how long does it take to give you the first page of results from:

```
find /usr | less
```

Example 3: Monte Carlo simulation

Here is a Monte Carlo estimator for the value of π (3.14159...) written as a generator function:

```
import random  
  
def estimate_pi_gen():  
    k = 0; n = 0  
    while True:  
        x = random.random()  
        y = random.random()
```

41

```
if x**2 + y**2 < 1:  
    k += 1  
    n += 1  
    yield k / n * 4
```

```
pi = estimate_pi_gen()
```

42

```
type(pi)
```

43

generator

```
next(pi)
```

136

3.010752688172043

```
# Generate 100 consecutive estimates from the 10000th:  
estimates = list(islice(pi, 10000, 10100))
```

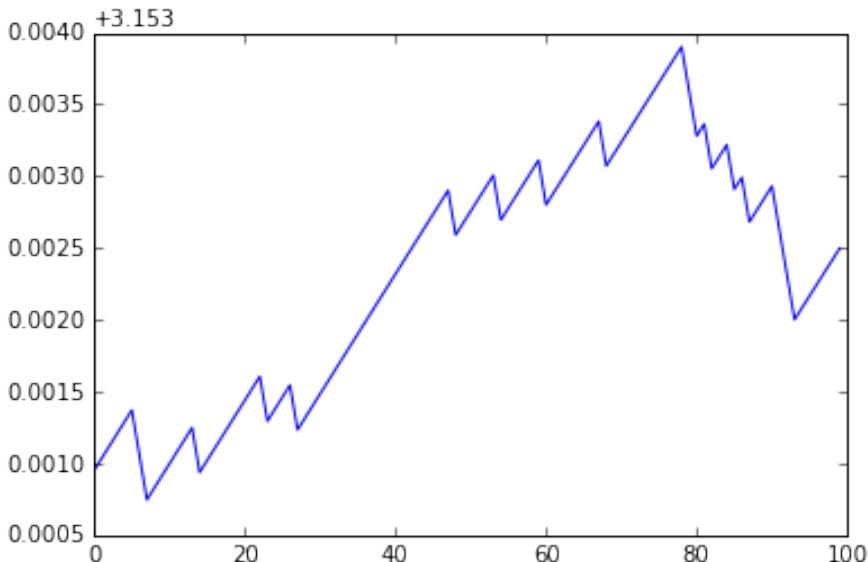
137

```
# Plot them:  
%matplotlib inline  
import matplotlib.pyplot as plt
```

138

```
plt.plot(estimates);
```

139



Exercise 2. Use `os.walk` and two one-line generator expressions to generate all `.txt` files on your disk lazily.

```
# See solutions/gen_ex2.py
```

1

Exercise 3: Blackjack simulation (requires classes)

(1a) Define a `Card` class for playing cards.

Spec: Initialize cards with a suit (clubs, diamonds, hearts, spades) and a rank (A, 2-10, J, Q, K).

(1b): Define a method `is_royal()` that returns `True` or `False`.

(1c): Add a `bj_value()` method, which returns the card's value in Blackjack (counting royal cards as 10 and Ace as 1).

```
# See solutions/blackjack_simulation_1.py
```

1

(2a) Use `random.sample()` to draw 1 million random “hands” of 3 cards from a standard deck of 52 cards.

(2b) What proportion of blackjack hands with 3 cards go “bust” (sum of values > 21)?

```
# See solutions/blackjack_simulation_2.py
```

3

```
# See solutions/blackjack_simulation_3.py
```

2

3.6 Iterators

An **iterator** is an object that can be accessed element-by-element successively. All an iterator requires is a `__next__` method. Advancing an iterator is done by applying the `next()` function to an object.

Example: files are iterators

```
f = open('/data/ASX.csv')
'__next__' in dir(f)
```

1

True

```
line1 = next(f)
line2 = next(f)
```

2

```
line1
```

3

```
'Symbol,Exchange,Name,Months,Quandl Code\n'
```

```
line2
```

4

```
'AP,ASX,SPI 200 Index,HMUZ,ASX/AP\n'
```

Non-example: In Python 3, `range` objects are not iterators:

```
range(5)
```

5

```
range(0, 5)
```

```
'__next__' in dir(range(5))
```

6

```
False
```

```
next(range(5))
```

7

To recap, Python calls `iter()` on an object under the hood when you use the object in an iterable context. This in turn calls the `.__iter__()` method on the object, which returns an **iterator** (an object with a `__next__()` method).

```
thing = [1, 2]
```

8

```
# This simple syntax:  
# for a in thing:  
#     print(a)  
  
# ... is very similar to this:  
iterable = iter(thing)  
while True:  
    try:
```

10

```
a = next(iterable)
except StopIteration:
    break
print(a)
```

1
2

You can also call the builtin `iter()` function explicitly on an iterable object:

```
myrange = range(5, 10)

iterator1 = iter(myrange)
for i in iterator1:
    print(i)

iterator2 = iter(myrange)
for i in iterator1:
    print(i)
```

Here, although `iterator1` is “used up” by the first loop, `iterator2` is created afresh.

Extended exercise: your own iterator

Try creating your own iterator class for leap years.

Remember that all that is required for an iterator is a `__next__` method.

Task: Create an iterator returning each leap year since a given year. The iterator stops when a leap year is found beyond a given maximum.

Hint: you can use the `isleap` function from Python’s `calendar` module to help you:

```
# Helper function:
from calendar import isleap
isleap(2003), isleap(2004), isleap(2005)
```

11

```
(False, True, False)
```

```
class LeapYearIterator:  
    # TO DO: define your __init__ and __next__ methods here
```

12

To make `LeapYearIterator` objects iterable, also define a `__iter__` method that returns an iterator object – i.e. `self`.

Goal: Then you can use it like this:

```
# Year 2004 is a leap year.  
2004 in LeapYearIterator(1890)
```

13

True

```
# Year 1900 is not a leap year.  
1900 in LeapYearIterator(1890)
```

14

False

```
# Because __iter__ is defined, you can use LeapYearIterator  
# in a for loop.  
for leap_year in LeapYearIterator(1980):  
    print(leap_year)
```

15

```
1984  
1988  
1992  
1996  
2000  
2004  
2008  
2012  
2016  
2020
```

```
# Or you can generate a list of all leap years:
leap_years = list(LeapYearIterator(1980))
leap_years
```

16

[1984, 1988, 1992, 1996, 2000, 2004, 2008, 2012, 2016, 2020]

See `solutions/leap_year_iterator.py`.

Notice how the `LeapYearIterator` class above is more complex and difficult to get right than the generator versions above!

Example: working with lazy sequences

For example, this would count the words in even a 10 terabyte text file:

```
from itertools import chain
```

143

```
filename = '/data/alice_in_wonderland.txt'
line_by_line = (line.upper().split() for line in
    open(filename))
next(line_by_line)
```

144

["ALICE'S", 'ADVENTURES', 'IN', 'WONDERLAND']

```
# This flattens the iterator of lists of words into
# a single iterator:
words = chain(*line_by_line)
```

145

```
next(words)
```

146

'LEWIS'

```
next(words)
```

147

```
'CARROLL'
```

```
from collections import Counter
counts = Counter(words)
```

148

```
counts.most_common(3)
```

149

```
[('THE', 1604), ('AND', 766), ('TO', 706)]
```

3.7 Summary

- A **generator** is a lazy sequence. Its values are computed on-the-fly when iterated over.
- A **generator expression** is a one-liner for creating a generator. Example:

```
filesizes = (getsize(path) for path in paths if isfile(path))
```

- A **generator function** is a function that returns a generator when called. A function is a “generator function” if it contains the keyword `yield`.
- A generator is a special case of an **iterable** data type. An iterable data type is one that can return each of its items one at a time in an iterable context like a `for` loop.
- Writing a generator is usually easier than writing a custom iterable data type. To be iterable, an object only needs an `__iter__` method which returns an iterator. (Usually the `iter()` call is implicit from using the object in an iterable context like a loop.)
- An **iterator** is simply an object that has a `__next__` method which returns the next successive item when called and raises a `StopIteration` exception when there are no more items.

3.8 Exercises: streaming analytics

Python's lazy sequences can be used in streaming analytics applications in a similar way to functional programming languages and Java's Streams API.

These extended exercises walk you through doing this with Python generators and the powerful 3rd-party `toolz` package.

Exercise 1: use `toolz` to sum up the total file sizes by extension in a folder recursively

Tips: 1. Use `p = pathlib.Path('/Data')` and `p.glob('**/*')` 2. Define these helper functions:

```
def get_ext(path):
    return path.suffix

def get_filesize(path):
    return path.lstat().st_size
```

18

Part (a):

1. Use `toolz.groupby` to create a dictionary mapping file extensions to lists of `Path` objects.
2. Define a function `sum_filesizes(paths)` using `get_filesize(path)`.
3. Use a `for` loop to sum up the total file size for each file extension.
4. Use a dict comprehension to do the same as above in 2.

Part (b):

5. Try to achieve the same effect as `sum_filesizes` using `map`, `toolz.partial`, `sum`, and `toolz.compose`. Is this easier?
6. Try to achieve the same effect as `sum_filesizes` using `pipe` and `toolz.curried.map`. Is this easier?
7. Use `toolz.valmap` on the output of ex1 to sum up your file sizes for each extension. Is this easier than ex2 and ex3?

Part (c): Do the same as before but using less memory. (The output of `groupby` had concrete lists.)

8. Define an `add(total, path)` function that looks up the file size for `path` and returns this added to the interim `total`
9. Use `toolz.reduceby` with `add` to answer the question.

See `solutions/total_filesize_by_ext.py`

Exercise 2: word count

Task: 1. Implement a function `wordcount(filename)` which returns a dict of counts of word in that file.

Tips:

- (a) use `open(..., mode='rb')` to sidestep text encoding problems.
 - (b) Check out `tz.frequencies`, `tz.concat`, `tz.compose`, and `tz.pipe`.
2. Implement a map-reduce style function `wordcount_all(folder)` that combines the word-count dictionaries. (Try `toolz.merge_with`).
 3. What are the top 20 words from all files in `/Data` and their counts?

Tip: see `tz.topk`

See `solutions/streaming_wordcount.py`

Chapter 4

Intermediate Python language features

This chapter introduces language features that various Python libraries (like `flask`, `requests`, `pandas`, and `numba`) make heavy use of. Understanding how they work will help you to use such libraries; they will also help you write your own Python code that makes full use of what the language has to offer.

4.1 Decorators

As context, Python has most features of functional programming languages. This includes *first-class functions*, which means that Python can manipulate functions in another function.

Using decorator functions allows a form of “meta-programming”, enabling you to add additional functionality to an existing function, method or class without touching its code.

A “decorator function” is a function that takes a function or method as an argument and returns a new function with additional functionality added to it (“decorated”).

In fact, you can define a decorator for any “callable” – a function, method or even a class, but for now we will only consider a decorator for a function.

Generally, you’ll write a few decorators and apply them many times.

Example 1

Recall that a decorator is a function that takes a function and returns a different function. Here is a simple example:

```
def make_bold(func):  
    def inner():  
        return '<b>' + func() + '</b>'  
    return inner
```

240

```
def hello():  
    return 'Hello!'
```

241

```
hello_bold = make_bold(hello)
```

242

```
hello_bold()
```

243

```
'<b>Hello!</b>'
```

Decorator syntax

Python supports special syntax for applying a decorator to a function when you define it:

```
@make_bold  
def hello():  
    return 'Hello'
```

245

```
hello()
```

246

```
'<b>Hello</b>'
```

```
# Equivalent to:  
def hello():  
    return 'Hello'  
hello = make_bold(hello)
```

247

Exercise:

1. change your `hello` function to take an argument (e.g. a person's name).
2. try it
3. to make this work, you'll need to change `inner` to take an argument.

Example 2

You can define decorators with custom behaviour – including parameters. Here is an example: let's define a decorator to check whether or not a function returns a value within given bounds:

```
# Example use:  
  
@bounded(3, 4)  
def square(x):  
    return x**2
```

1

```
# Would work:  
# square(3.5)
```

1

```
# Exception:  
# square(10)
```

1

Exercise: try implementing this

Hint: you'll need a function that returns a decorator with the same structure as `make_bold`

Solution

See `solutions/bounded_decorator.py`

Exercise - `timeme` decorator

Write a decorator called `timeme` which times how long a function takes to run and prints this to `stdout`.

```
# Hint:  
  
import time  
  
time.time() # number of seconds since the Epoch
```

279

```
# Example use:  
@timeme  
def slow(n=10**8):  
    return sum(range(n))
```

1

```
# Calling `slow()` should return the result and print how  
# long the  
# function took to run.  
# Example:  
slow()
```

1

```
Function took 1.350 seconds
4999999950000000
# See solutions/timeme.py
```

Example packages that use decorators:

- Flask for web development
- Dash for web dashboards
- Python's built-in decorators: `property`, `classmethod`, `staticmethod`, ...

4.2 Context managers

Context managers are useful when you want to create an object that automatically initializes itself before executing a code block and cleans up itself after executing the code.

In short, context managers help with managing global state before and after a block of code executes.

First, here is an example of writing text to a file using `try` and `finally` manually:

```
file = open('new_file.txt', mode='wt')
try:
    file.write('data goes here')
finally:
    file.close()
```

1

In this code, `finally` makes sure to close the open file, even in a case an exception is raised. The `with` statement offers a more elegant way of expressing this:

```
with open('new_file.txt', 'wt') as f:
    f.write('data')
```

1

The `with` statement handles opening the file and returning the handle, but most importantly, if there is an error when you're writing this file the context handles the exception and makes sure the file is saved properly, and data is written to the file as expected.

A context manager is an object to be used in a `with` statement. A file object is one common example of a context manager. All context managers creates their own contexts using methods called `__enter__` and `__exit__`:

```
with open('new_file.txt', mode='rt') as file:
    print(dir(file))
```

34

```
['__CHUNK_SIZE', '__class__', '__delattr__', '__dict__',
 '__dir__', '__doc__', '__enter__', '__eq__', '__exit__',
 '__format__', '__ge__', '__getattribute__',
 '__getstate__', '__gt__', '__hash__', '__init__',
 '__iter__', '__le__', '__lt__', '__ne__', '__new__',
 '__next__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', '_checkClosed', '_checkReadable',
 '_checkSeekable', '_checkWritable', 'buffer', 'close',
 'closed', 'detach', 'encoding', 'errors', 'fileno',
 'flush', 'isatty', 'line_buffering', 'mode', 'name',
 'newlines', 'read', 'readable', 'readline', 'readlines',
 'seek', 'seekable', 'tell', 'truncate', 'writable',
 'write', 'writelines']
```

When a file is opened using the `with` statement, the file object invokes its `__enter__` method. After executing the code within `with` statement, the file object invokes the `__exit__` method, which closes the open file.

To check these method invocations, let's create our own simple context manager.

```
class CM(object):
    def __init__(self):
        print("I love Python.")

    def __enter__(self):
```

35

```
print("__enter__")
return self

def __exit__(self, exc_type, exc_val, exc_tb):
    print("__exit__")
```

```
with CM() as obj:
    obj.do()
```

36

```
__enter__
I love Python.
__exit__
```

As you can see, when a context manager object is created with the `with` statement, `__enter__` and `__exit__` methods are automatically called to execute a context for the object.

4.3 Easily creating context managers

The same pattern applies when using other context managers – code is executed when the block is opened, potentially providing an object, and the executes code when the block closes. It is easy to write your own context manager that follows this pattern. You will use the `contextlib.contextmanager` decorator around a function which will `yield` a value. Everything in front of the `yield` will be executed before the start of the block (in `__enter__`); everything afterwards will execute at the close of the block (in `__exit__`).

Consider working with Pandas DataFrames: it is sometimes useful to be able to see every single value in the DataFrame all at once. It is easy to set the display limits of DataFrames by setting `max_rows = None` and `max_columns = None` from `pandas.options.display`, but it's likely you'll only want to do this occasionally. So you can make a context manager that will handle the display, and reset it back to the original value on exit.

First, read some data, and set some usefully small values for `max_rows` and

`max_columns` to provide snapshots of the data:

```
import pandas as pd

pd.options.display.max_rows = 5
pd.options.display.max_columns = 5

olympics = pd.read_csv('Data/olympics2012.csv')
olympics
```

	Country	Gold	Silver	Bronze
0	Afghanistan	0	0	1
1	Albania	0	0	0
...
202	Zambia	0	0	0
203	Zimbabwe	0	0	0

Then write the context manager as follows:

```
from contextlib import contextmanager

@contextmanager
def display_full_dataframe():
    # This code is run before the context block:
    old_max_rows = pd.options.display.max_rows
    old_max_columns = pd.options.display.max_columns
    pd.options.display.max_rows = None
    pd.options.display.max_columns = None

    yield

    # This code is run after the context block:
    pd.options.display.max_rows = old_max_rows
    pd.options.display.max_columns = old_max_columns
```

When working from within Jupyter notebook, we'd need to change the following setting so that the DataFrame's is displayed even from within the context:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = 'last'
```

25

Now:

```
with display_full_dataframe():
    olympics[:10]
```

29

	Country	Gold	Silver	Bronze
0	Afghanistan	0	0	1
1	Albania	0	0	0
2	Algeria	1	0	0
3	American Virgin Islands	0	0	0
4	Andorra	0	0	0
5	Angola	0	0	0
6	Antigua and Barbuda	0	0	0
7	Argentina	1	1	2
8	Armenia	0	1	2
9	Aruba	0	0	0

After with with block, the 5-row limit is reinstated:

```
olympics[:10]
```

30

	Country	Gold	Silver	Bronze
0	Afghanistan	0	0	1
1	Albania	0	0	0
...
8	Armenia	0	1	2
9	Aruba	0	0	0

Exercise: Write a context manager to temporarily change the working directory

Following the pattern above write your own context manager called `move_directory` to change directory for the duration of the with statement. Look at `os.chdir` and `os.getcwd` for changing the Python process' working environment.

Test this with:

```
with move_directory('Data'):
    forbes = pd.read_csv('forbes1964.csv')
    with display_full_dataframe():
        forbes
```

1

4.4 Extended exercise: Beyond PEP8

Raymond Hettinger, one of Python's core developers, gave a talk at PyCon 2015 entitled "Beyond PEP8 – Best practices for beautiful intelligible code". This section examines the code example he gave in that talk in detail. (If you would like to watch the talk, the link is here.)

The starting point is this code, which has been transliterated from Java into Python. The code is not particularly "Pythonic". It loops over network elements in a routing table and prints them out:

```
# See extras/rt_ugly.py
import jnettool.tools.elements
from jnettool.tools import Routing
from jnettool.tools import RouteInspector

ne=jnettool.tools.elements.NetworkElement( '171.0.2.45' )
try:
    routing_table=ne.getRoutingTable()  # fetch table

except jnettool.tools.elements.MissingVar:
    # Record table fault
    logging.exception( '''No routing table found''' )
    # Undo partial changes
    ne.cleanup( '''rollback''' )

else:
    num_routes=routing_table.getSize()      # determine table
    ↵ size
    for RToffset in range( num_routes ):
```

1

```

        route=routing_table.getRouteByIndex( RToffset )
        name=route.getName()          # route name
        ipaddr=route.getIPAddr()       # ip address
        print("%15s -> %s" % (name,ipaddr)) # format nicely
finally:
    ne.cleanup( '''commit''' ) # lockin changes
    ne.disconnect()

```

According to Hettinger, “well-written Python code looks like business logic.”

Exercise: make this Pythonic

We walk through an exercise of transforming the API for this code step-by-step to make it more Pythonic. The goal is to write supporting logic so the above code can be written like this:

```

# See extras/rt_elegant.py
from nettools import NetworkElement

with NetworkElement('171.0.2.45') as ne:
    for route in ne.routing_table:
        print("%15s -> %s" % (route.name, route.ipaddr))

```

1

Step 1: Create a single `nettools.py` module that uses the “adapter pattern” to wrap the existing `jnettool` package. (You will not reimplement the business logic, merely wrap the old API to the new API.)

Step 2: Create custom exceptions with clear names.

Step 3: Define `Route`, `RoutingTable`, and `NetworkElement` as classes. Add properties instead of the getter methods `getRoutingTable`, `getSize`, `getName`, and `getIPAddr`.

Step 4: Create a context manager for recurring setup and teardown logic. This should implement the exception-handling above.

Step 5: Use magic methods:

- `__len__` instead of `getSize()`

- `__getitem__` instead of `getRouteByIndex()`
- make the table iterable

Step 6: Add a good `__repr__` for easier debugging.

Chapter 5

Managing memory

5.1 Variable names are just labels

To start, note that variables in Python are just names or labels for objects. Think of them as post-it notes on objects instead of as a predefined box that you put data into, as with statically typed languages:



```
a = 'Me'  
b = a  
a = 'Myself'  
a, b
```

56

```
('Myself', 'Me')
```

5.2 Mutability

Python offers both mutable and immutable data types. Those that are “immutable” cannot be changed once created.

Mutable data types offer the advantage of convenience of use in imperative programming constructs like `for` loops.

Immutable data types offer the advantage of safety; an immutable object cannot be corrupted by passing it into a badly written function. Immutable objects are also inherently thread-safe; there is no need to lock something

that cannot change. The immutability also makes it possible to share immutable objects more cheaply.

There are performance advantages and disadvantages to mutability. Operations that mutate an object, like string concatenation, can sometimes be done with less memory copying with mutable types. On the other hand, operations that share an object, like passing it to a function, benefit from reduced memory consumption and improved cache utilisation with immutable types.

Perhaps more important than performance is the difference in programming style that immutable types enforces: safer and more functional, and hence more parallelizable.

5.3 Mutable and immutable types

Immutable	Mutable
tuple	list
bytes	bytearray
str	
frozenset	set
frozendict	dict
range	
int	
float	
complex	

Alternatives for a mutable string include a `list` of strings and `io.StringIO`, which is an in-memory file-like stream for text IO.

```
s = frozenset(['a', 'b', 'a', 'c'])
```

5

```
print(dir(s))
```

8

```
[ '__and__', '__class__', '__contains__', '__delattr__',
  __dir__, '__doc__', '__eq__', '__format__', '__ge__',
  __getattribute__, '__gt__', '__hash__', '__init__',
  __iter__, '__le__', '__len__', '__lt__', '__ne__',
  __new__, '__or__', '__rand__', '__reduce__',
  __reduce_ex__, '__repr__', '__ror__', '__rsub__',
  __rxor__, '__setattr__', '__sizeof__', '__str__',
  '__sub__', '__subclasshook__', '__xor__', 'copy',
  'difference', 'intersection', 'isdisjoint', 'issubset',
  'issuperset', 'symmetric_difference', 'union']
```

This raises a `TypeError`:

```
del s['a']
```

9

5.4 object `id`

The `id()` function returns a unique numerical ID for each named object. If two variable names have the same `id()`, they refer to the same object:

```
a = 1
id(a)
```

1

4297327232

```
b = a
```

2

```
id(b)
```

4

4297327232

```
def double(myvar):
    print(id(myvar))
    myvar *= 2
    print(id(myvar))
```

7

```
double(a)
```

8

```
4297327232  
4297327264
```

```
id(a)
```

9

```
4297327232
```

The `id` of `a` has not changed. This is because `a` is an integer and integers are immutable.

Contrast what happens when we pass a mutable object like a list into `double`:

```
names = ['Barney', 'Betty', 'Bambam']
```

18

```
double(names)
```

19

```
4394620744  
4394620744
```

```
names
```

20

```
['Barney', 'Betty', 'Bambam', 'Barney', 'Betty', 'Bambam']
```

Strings are also immutable:

```
s = 'abc'  
id(s)
```

12

```
4338148216
```

```
s += 'def'  
id(s)
```

13

4395430328

You see the above has actually sneakily created a new string object `s`. The ID has changed.

Because integers and strings are immutable (see the table above), Python “fakes” operations like `+=` by reassigning the variable name to a new object with the new value. Another example:

```
a += 1  
a
```

25

2

```
id(a)
```

26

4297326656

```
id(a) == id(b)
```

28

`False`

`a` refers to a different object now (2), but `b` still refers to the same one (1).

```
b
```

29

1

```
id(b)
```

20

4297326624

The same is true with strings:

```
s = 'Hello'  
id(s)
```

30

4416281768

```
s += ' world'  
id(s)
```

31

4416185840

This is why string concatenation is relatively slow in Python and why `str.join()` is recommended instead.

Comparison with other languages

The following languages have immutable strings: - Python - Java - C# - Haskell - Clojure - Scala

5.5 Pass-by-reference or pass-by-value?

Python passes both immutable and mutable objects by reference, but not in the same sense as in Java. Passing a variable has the effect as passing the `ids` of the objects.

```
i = 1  
s = 'Hello'  
l = ['Item 1', 'Item 2']  
  
def f(i, s, l):  
    return (i, s, l)  
  
(i2, s2, l2) = f(i, s, l)  
  
print('Same object?')  
print('i:', id(i) == id(i2))
```

39

```
print('s:', id(s) == id(s2))
print('l:', id(l) == id(l2))
```

Same object?

```
i: True
s: True
l: True
```

Question

Is it possible to write a `swap(a, b)` function in Python that swaps the values of `a` and `b` in-place?

Answer

Yes, but only if `a` and `b` are mutable.

Exercise

1. Write such a function for lists.

```
a, b = b, a
```

1

One solution:

```
def swap(a, b):
    c = a.copy()
    d = b
    a[:] = b
    b[:] = c
```

34

```
names1 = ['Fred', 'Betty']
print(id(names1))
names2 = ['Wilma', 'Barney']
print(id(names2))
```

35

```
4395242248  
4395461960
```

```
swap(names1, names2)
```

36

```
names1
```

37

```
['Wilma', 'Barney']
```

```
names2
```

38

```
['Fred', 'Betty']
```

2. Try writing such a function for tuples.

Exercise: Can you explain this?

```
def reassign(list):  
    list = [0, 1]  
  
list = [0]  
reassign(list)  
print(list)
```

43

```
[0]
```

The line `list = [0, 1]` inside the function creates a local variable (name) which shadows the global one within the scope of the function.

Try printing the `id` of `list` outside and inside the function.

```
def append(list):  
    list.append(1)  
  
append(list)  
print(list)
```

44

```
[0, 1]
```

5.6 Garbage collection

Python’s memory is automatically allocated and freed.

The garbage-collection implementation varies among different Python interpreters (CPython, PyPy, ...).

The CPython interpreter uses both reference-counting and scheduled garbage collection. When no more references point to an object, its memory is freed. This is simple and efficient, provided that there are no “reference cycles” (two objects containing references to each other). To the extent possible, it is good practice to write code without reference cycles.

CPython’s scheduled garbage collector takes time to run, so CPython schedules garbage collection periodically – when the difference between the number of allocations and deallocations exceeds a threshold:

```
import gc  
gc.get_threshold()
```

61

```
(700, 10, 10)
```

The threshold here is 700 allocations. (The three values refer to thresholds for 3 generations. See the `gc` docs for more details.)

Manual garbage collection

```
import gc  
gc.collect()
```

62

335

The value returned is the number of objects it has collected and deallocated.

For full details of CPython’s garbage collector, see: <http://hg.python.org/cpython/file/tip/gcmodule.c>.

When to use manual garbage collection

If you are seeing memory leaks with long-running server processes, try invoking `gc.collect()` periodically.

If responsiveness is a problem (e.g. for a server that must respond quickly to requests), you can turn off garbage collection with `gc.disable()` and then either:

- manually collect when the system is idle
- configure `mod_wsgi` to kill and restart processes more frequently.

Garbage collection statistics

To find out more, use:

```
gc.set_debug(gc.DEBUG_STATS)
```

63

You can also find places that require attention with:

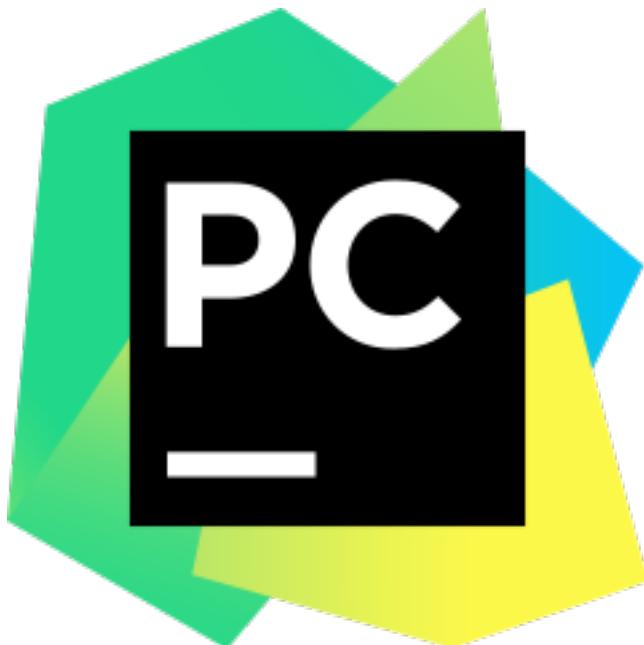
```
a = []
b = [a]
a.append(a)
referrers = gc.get_referrers(a)
```

67

Chapter 6

Introduction to PyCharm

PyCharm developed by JetBrains is a professional Integrated Development Environment (IDE) for Python developers. It is extremely powerful, flexible and customisable, but the first time you use PyCharm, it can be a little intimidating to get set up.



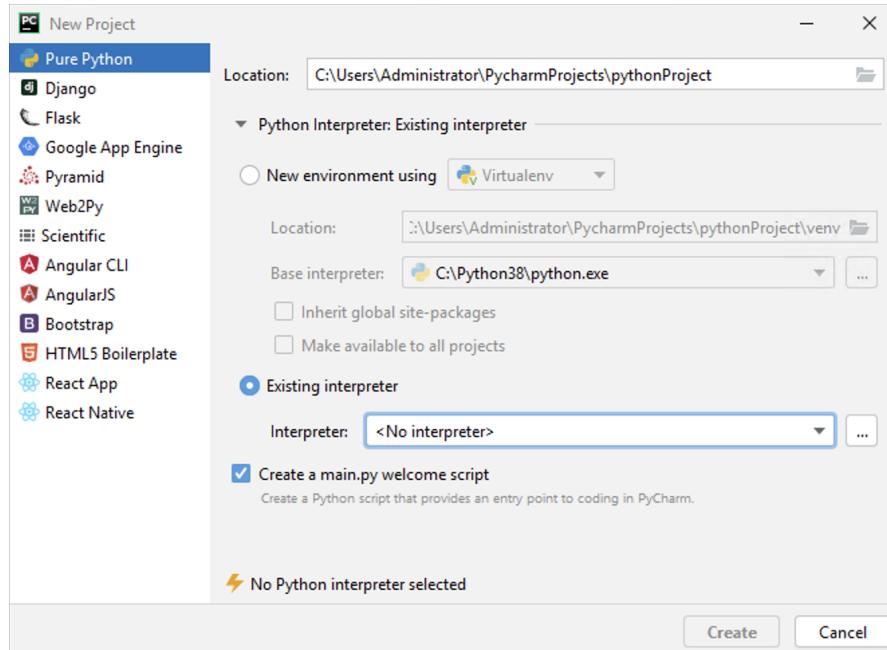
For the purpose of this chapter it's assumed that you have the Professional version of PyCharm installed.

6.1 Creating a new project in PyCharm

In PyCharm creation of a new project has a few steps. The first is very simple: select New Project from the PyCharm welcome screen.



In the new project screen select the kind of project you wish to create. Using the professional version of PyCharm you will have a much wider set of choices available to you, but for your first project choose a Pure Python project.



In the Location bar navigate to the folder where you want to create your project. Alternately you can browse to a folder that already contains a project (you will be asked if you want to create a project from existing sources - say “Create from Existing Sources”).

A good place to start is by having a standard project you can work from. Creating a Python library is covered elsewhere in the notes, but for now we'll use the Cookiecutter library to create a project automatically. In a terminal prompt, navigate to the folder you want to contain your project and:

```
pip install -U cookiecutter  
cookiecutter gh:PythonCharmers/cookiecutter-pipproject
```

1

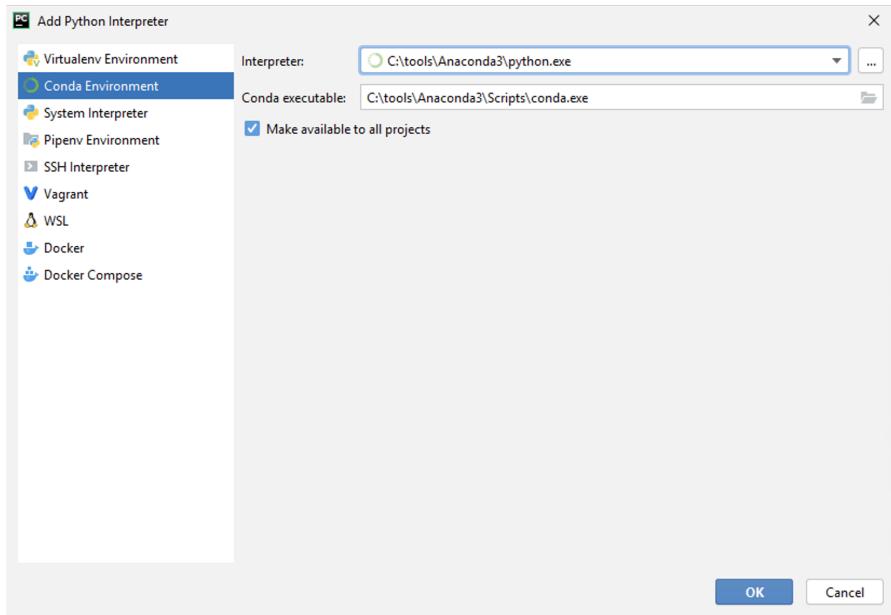
Answer the questions and you can create a project automatically. In these notes we're assuming you've created a bankaccount project.

6.2 The interpreter

Each PyCharm project should have an interpreter. PyCharm uses this to determine code completion, to find documentation, and to validate code.

As you get used to more development work in Python you can look at creating virtual environments on a project-by-project basis. For now you can assume that you are using a single Python environment for all projects.

First select that you want to use an existing interpreter. The first time you are using PyCharm you won't have any interpreters in the dropdown to choose from so select the ellipsis ... to set up a new interpreter.

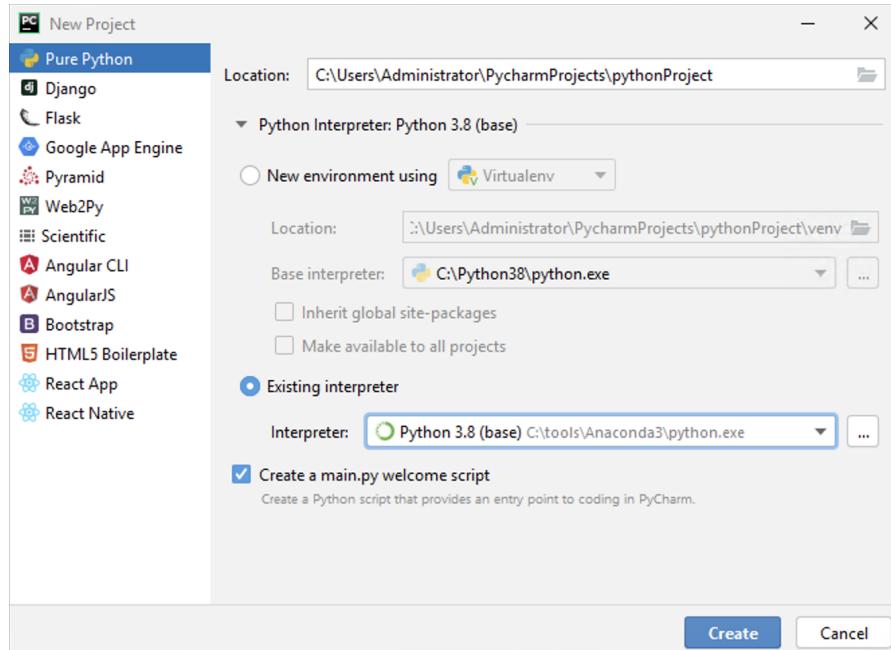


Assuming you are using Anaconda as your primary Python version select “Conda Environment” from the choices on the left. Note that in the professional version of PyCharm you will have many choices of interpreter that you won’t see using the open source / education version.

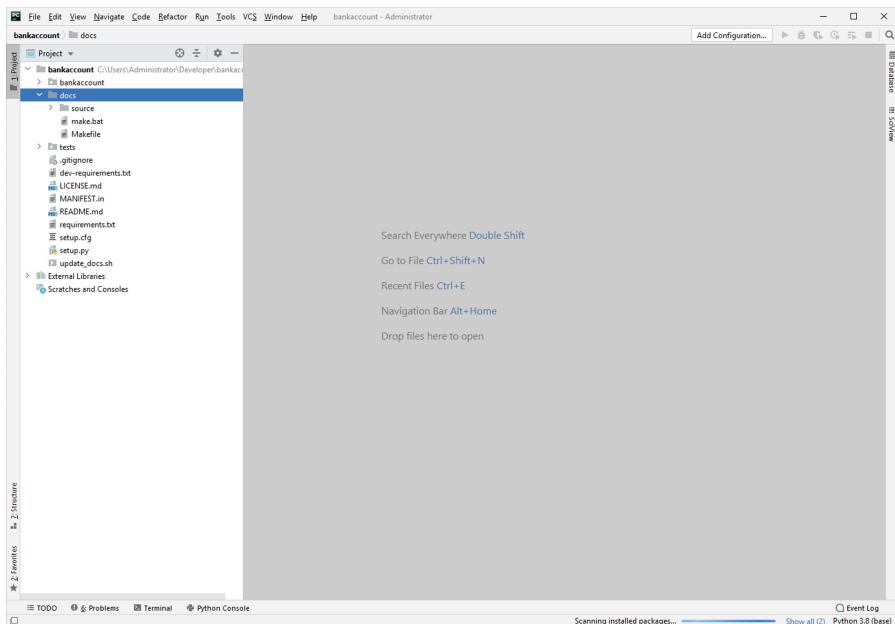
Once you have selected the “Conda Environment” locate the Anaconda installed `python.exe` (usually directly inside the folder where you installed Anaconda) as the Interpreter, and `conda.exe` (in the `Scripts` folder where you installed Anaconda) as the Conda executable.

Click OK to finish adding the new interpreter.

Click “Create” to create the new PyCharm project.



This opens the standard project view. Note that it may take a couple of minutes for PyCharm to index your installed libraries so that code completion is enabled. It will only have to do this the first time you use a particular Python interpreter, and it will index new libraries as they are installed.



6.3 Editing code

To start editing a file double click it in the Project tab on the left and it will open the file for you to edit. You will find Python files with a .py extension have code highlighting enabled along side PyCharm's internal code quality checks.

For example opening the `setup.py` will show you the content of the file and in the top left-hand corner you will see the results of the code quality checks allowing you to step through errors, warnings, and even spelling mistakes.

```

bankaccount  setup.py
Project  S: 2 projects
  bankaccount  /Developer/work/PythonCharmers/bankacc...
    >  bankaccount
      ->  __init__.py
      ->  bankaccount
        ->  __init__.py
        ->  source
          ->  make.bat
          ->  Makefile
        ->  .gitignore
        ->  .gitattributes
        ->  LICENSE.md
        ->  MANIFEST.in
        ->  README.md
        ->  requirements.txt
        ->  setup.py
        ->  setup.py
        ->  update_docs.sh
      ->  External Libraries
    ->  Scratches and Consoles

  S: 2 projects
  Problems  Terminal  Python Console  TODO  Event Log  33:01 LF  UTF-8  6 spaces  Python 3.7 [selected]  Run  Help

  bankaccount -> setup.py
  setup()
  -----
  1   from setuptools import setup, find_packages
  2   from codecs import open
  3   from os import path
  4
  5   ...VERSION... = "0.0.1"
  6
  7   here = path.abspath(path.dirname(__file__))
  8
  9   # long description
 10  # Read long description from the README file
 11  with open(path.join(here, 'README.md'), encoding='utf-8') as f:
 12      long_description = f.read()
 13
 14  # dependencies
 15  # Define your dependencies and installs
 16  # with requirements.txt
 17  with open(path.join(here, 'requirements.txt'), encoding='utf-8') as f:
 18      all_reqs = f.read().split('\n')
 19
 20  install_requires = [x.strip() for x in all_reqs]
 21
 22  # the actual setup_ call done through the setup() function
 23  setup(
 24      name='bankaccount',
 25      version=...VERSION...,
 26
 27      # List of dependencies created above (from requirements.txt)
 28      install_requires=install_requires,
 29
 30      # descriptions (including from README and the long_description)
 31      description='A tritrary package demonstrating the PyCharm project.',
 32      long_description=long_description,
 33      long_description_content_type='text/markdown',  # for README.md as markdown
 34
 35      # define what redistributions you give for others to use (what license)
 36      license='MIT',
 37
 38      # files etc you distribute in root folder except docs and tests folders
 39      packages=find_packages(exclude=['docs', 'tests']),
 40
 41      # include additional files in MANIFEST.in
 42      include_package_data=True,
 43      author='Henry Reithner',
 44      )
 45
 46

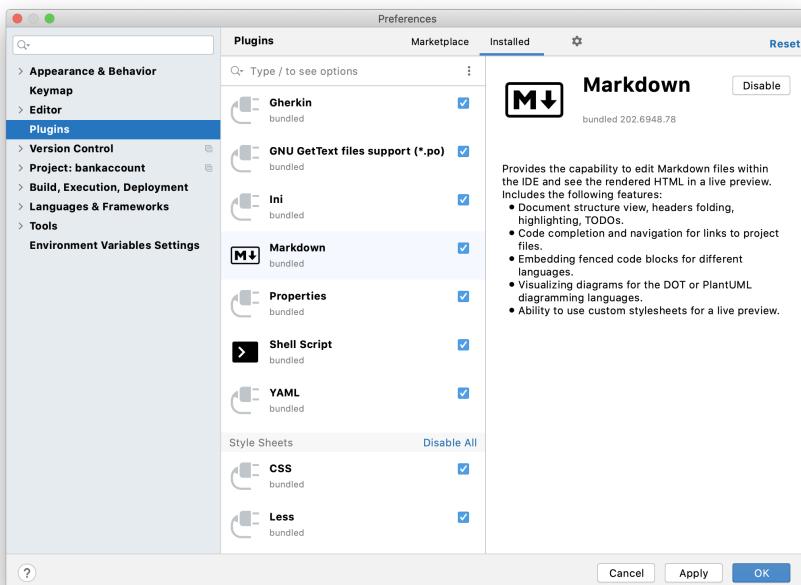
```

PyCharm extensions

If you open a new file type in PyCharm you may get prompted to install an extension. For example if you open the README.md file in your project you will be prompted to install the Markdown extension. Usually these extensions are very useful for helping to edit a variety of file formats.

On installing the extension you will be prompted to restart PyCharm. Do so, and the extension will be enabled for you to use in editing the file.

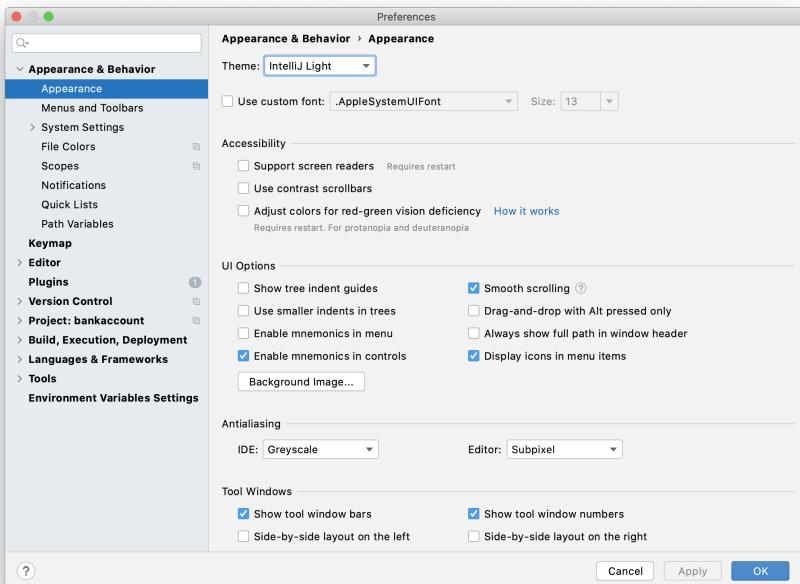
To see the extensions you have installed you can access them through the project preferences (from the File menu) under the plugins tab.



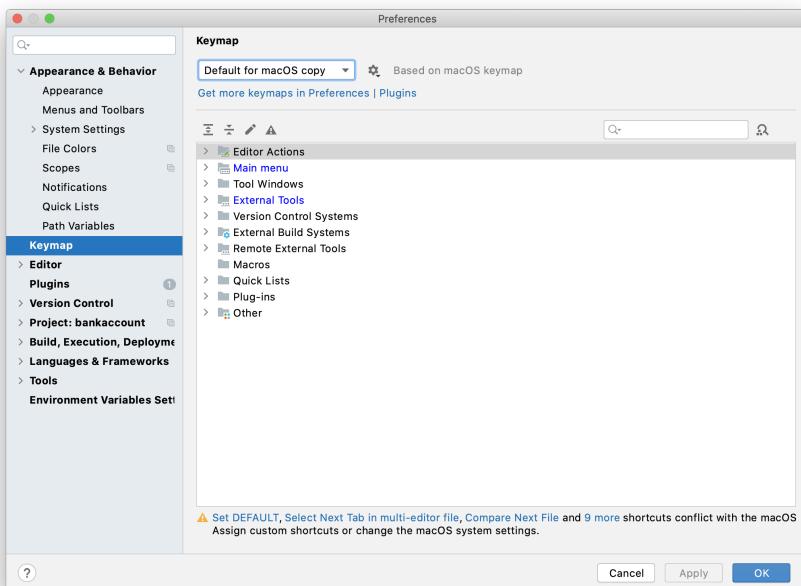
Customising PyCharm's look and feel

You have a great deal of control over the way that PyCharm works. When you first opened PyCharm you would have been prompted to choose a visual theme (light or dark) and to choose a keyboard shortcut scheme. These are not set in stone and can be changed through the project preferences (from the File menu).

The theme and other appearance components can be changed through the Appearance tab:

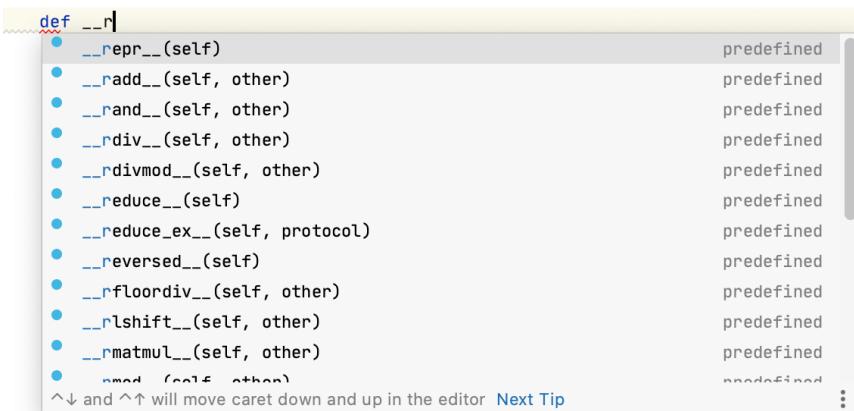


Likewise the keyboard shortcuts can be changed through the Keymap tab:



Code completion

When writing code PyCharm will infer data types from functions and class signatures and offer code completion based on this inferred type. By pausing typing PyCharm will popup code completion, or it can be activated with `Ctrl` + `Space` (On both macOS and Windows).



To navigate the code-completion you can use the mouse or \uparrow or \downarrow and to select an option click with the mouse or **Enter** on the keyboard.

Other keyboard shortcuts

There are other standard keyboard shortcuts it can be extremely useful to learn in PyCharm, which are dependent on your operating system:

macOS Shortcut	Windows Shortcut	Action
Double Shift	Double Shift	Search Everywhere. Find anything related to PyCharm or your project and open it, execute it, or jump to it.
Shift Command A	Ctrl + Shift + A	Find Action. Find a command and execute it, open a tool window or search for a setting.

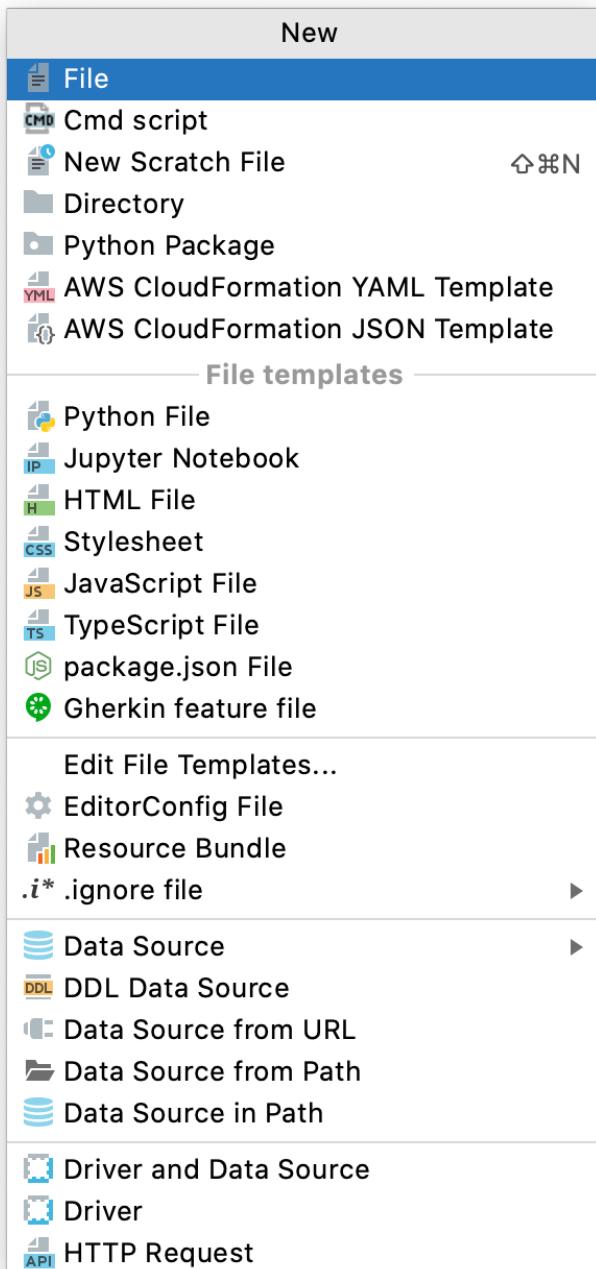
macOS Shortcut	Windows Shortcut	Action
Double Ctrl	Double Ctrl	Run Anything. Execute commands, such as opening a project, launching a run/debug configuration, running a command-line utility, and so on. The available commands depend on the set of plugins and tools you have configured for your project.
Option Enter	Alt Enter	Show intention actions and quick-fixes. Fix highlighted error or warning, improve or optimise a code construct.
F2 or Shift F2	F2 or Shift F2	Navigate between code issues. Jump to the next or previous highlighted error.
Command E	Ctrl E	View recent files. Select a recently opened file from the list.
Option ↑ or Option ↓	Ctrl W or Ctrl Shift W	Extend or shrink selection. Increase or decrease the scope of selection according to specific code constructs.

macOS Shortcut	Windows Shortcut	Action
Command / or Option Command /	Ctrl / or Ctrl Shift /	Add/remove line or block comment. Comment out a line or block of code.
Option F7	Alt F7	Find usages. Show all places where a code element is used across your project.

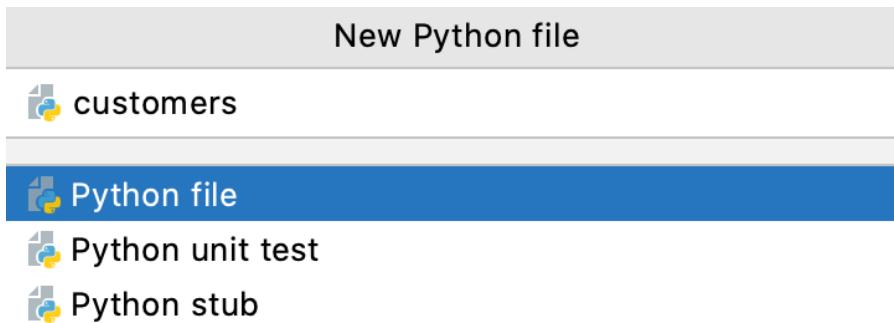
Creating new files in PyCharm

From the File menu select New, or use the keyboard shortcut Command N on macOS or Ctrl N, or right click in the file explorer and select the New sub-menu.

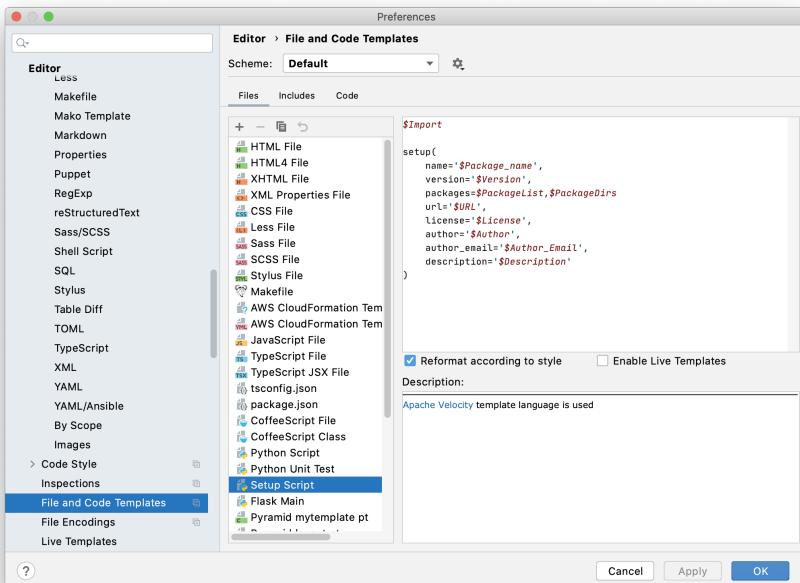
Regardless of how you access it you will be shown the New sub-menu:



To create a new file select either File, or from one of the many templates you have to choose from. By selecting a New Python File you will get a number of template stubs to choose from when creating your file:



You can customise PyCharm further with your own file templates (built on the Apache Velocity language) by opening the application preferences and selecting the File and Code Templates tab under the Editor group:



To add a template use the + button, name the template and select a file extension (e.g. py for a Python file). For example create a new template

called Pytest unittest and add the following code:

```
import $PackageName

import pytest

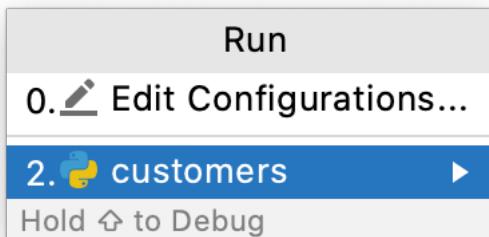
def test_mycode():
    """A simple unit test"""
    assert False
```

1

You will be able to select Pytest unittest from the New menu and be prompted to enter both the file name and the PackageName which will be automatically be populated into the file.

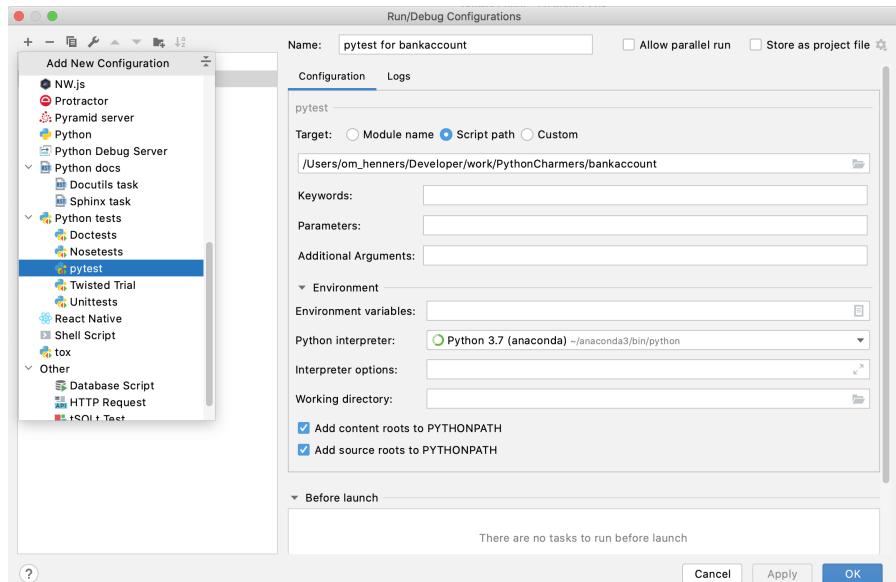
6.4 Running code in PyCharm

Running your code in PyCharm is easy. Selecting Run from the Run menu will prompt for a file to run (the file that you have open) or to edit run configurations.



If you select the currently opened file then a configuration will be automatically generated that runs that Python file with the current interpreter. This is usually the easiest method to start running your script, but of course it relies on your script doing something when run directly (usually via the `if __name__ == '__main__'` pattern).

If you're writing a Python package there's a good chance that you won't have a script entry point. This means you need to add a run configuration manually. Fortunately PyCharm has many run configuration templates available out of the box. Open the run configuration menu and select the + to add a new run configuration built from one of the built-in templates:



Select the desired template. In most cases once you have a test selected you will only be required to set up whether the configuration is running for a module or from a particular script path (folder).

Select OK and the run configuration will be listed with the name you provided in the run menu. **Note** despite unit tests being tests they do not have to be run as a debug operation - they still use the standard run configuration.

Any results of your run including output to the standard output (via log or print statements) or error messages will be visible in the Run tab at the bottom of the screen. Depending on the template that you selected there will be different output messages based on the code you ran but the principal is the same.

In the example a pytest run was completed that detected no errors in the code.

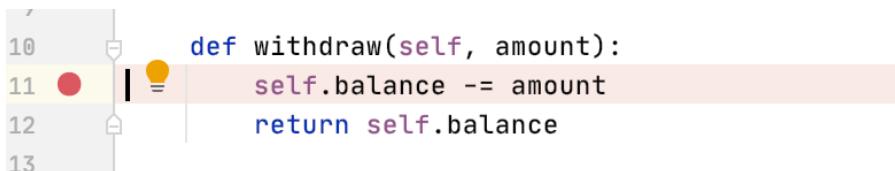
The screenshot shows the PyCharm interface with the 'Run' tool window open. The title bar says 'Run: pytest for bankaccount'. The 'Test Results' tab is selected, showing a single test named 'test_sample' that passed. The output pane displays the command run: 'python /Applications/PyCharm.app/Contents/plugins/python/helpers/runConfiguration/run_coverage.py run --omit= */Applications/PyCharm.app/Contents/plugins/python/helpers/testrunner/* --cov=bankaccount'. It also shows the Python version and other test-related details.

This makes unit testing your code very simple. If for example you have a unit test that fails and it is run with a unit test run template it will be shown in the run window as a failed test with a complete stack trace and the ability to jump to the code where there was an error.

The screenshot shows the PyCharm interface with the 'Run' tool window open. The title bar says 'Run: pytest for bankaccount'. The 'Test Results' tab is selected, showing a test named 'test_sample' that failed. The output pane displays the command run: 'python /Applications/PyCharm.app/Contents/plugins/python/helpers/runConfiguration/run_coverage.py run --omit= */Applications/PyCharm.app/Contents/plugins/python/helpers/testrunner/* --cov=bankaccount'. It includes the stack trace for the failure: 'tests/test_sample.py:10: AssertionError'.

6.5 Debugging in PyCharm

Debugging in PyCharm builds on existing run configuration. If you run the configuration in debug mode without doing anything else the results will be the same as a typical run. However the debug run allows you to set breakpoints in code. This is done by clicking in the margin of the editor on the line where you want the code to halt.



The next time you run your code in debug mode the code will halt at the breakpoint and you will be presented with the debug panel at the bottom of the screen.



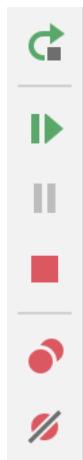
This presents you with the current stack on the left hand side and on the right the currently defined variables in the scope at the level you have selected. More importantly you are given debug controls at the top of the panel:



From left to right these are:

1. Step over to execute the current line and move onto the next line (staying in debug mode)
2. Step into to step down a further level of the stack and step through the code being called on that line
3. Step into my code is the same, but restricts you to only introspecting code in the module you're writing
4. Force step into which will step into the function regardless
5. Step out will resume running the code until it reaches another return or yield statement
6. Run to cursor resumes running the code until it reaches the cursor in the editor
7. Evaluate expression which lets execute arbitrary Python code

There are also controls down the left hand side which are most useful when combined with multiple breakpoints throughout your code:



From top to bottom these controls are:

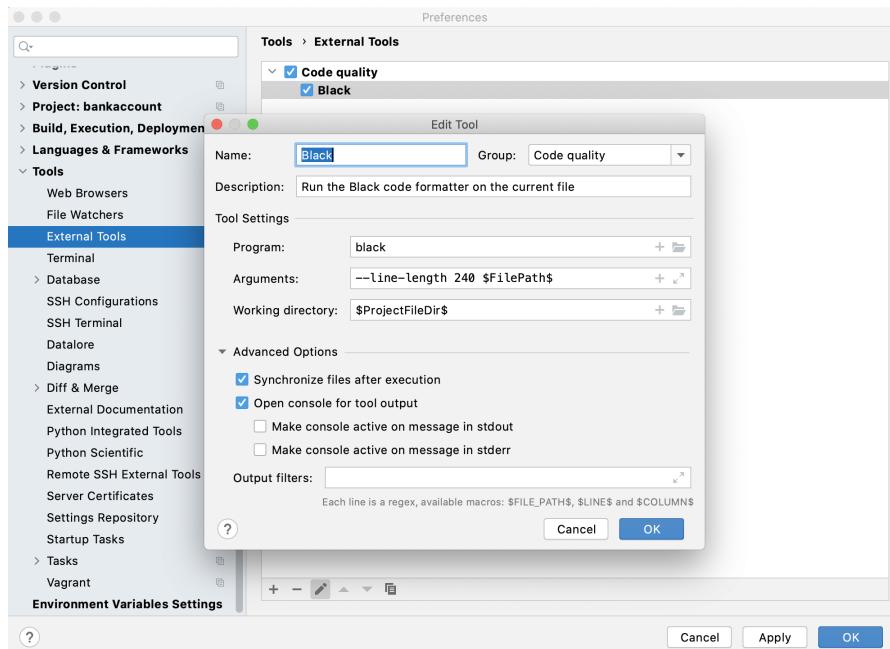
1. Stop and restart the execution of tests
2. Resume and keep processing code until another breakpoint is met
3. Pause the program
4. Stop the execution altogether
5. View a list of breakpoints in the code
6. Mute breakpoints to run while ignoring breaks

If you're familiar with the Python inbuilt debugging tool pdb these tools will be familiar to you, but even if you are not working with debugging tools in PyCharm is intuitive.

Using these controls you can walk through your Python code step by step to find out what's going wrong, change data values as the code is running, and confirm the code is working as you intended.

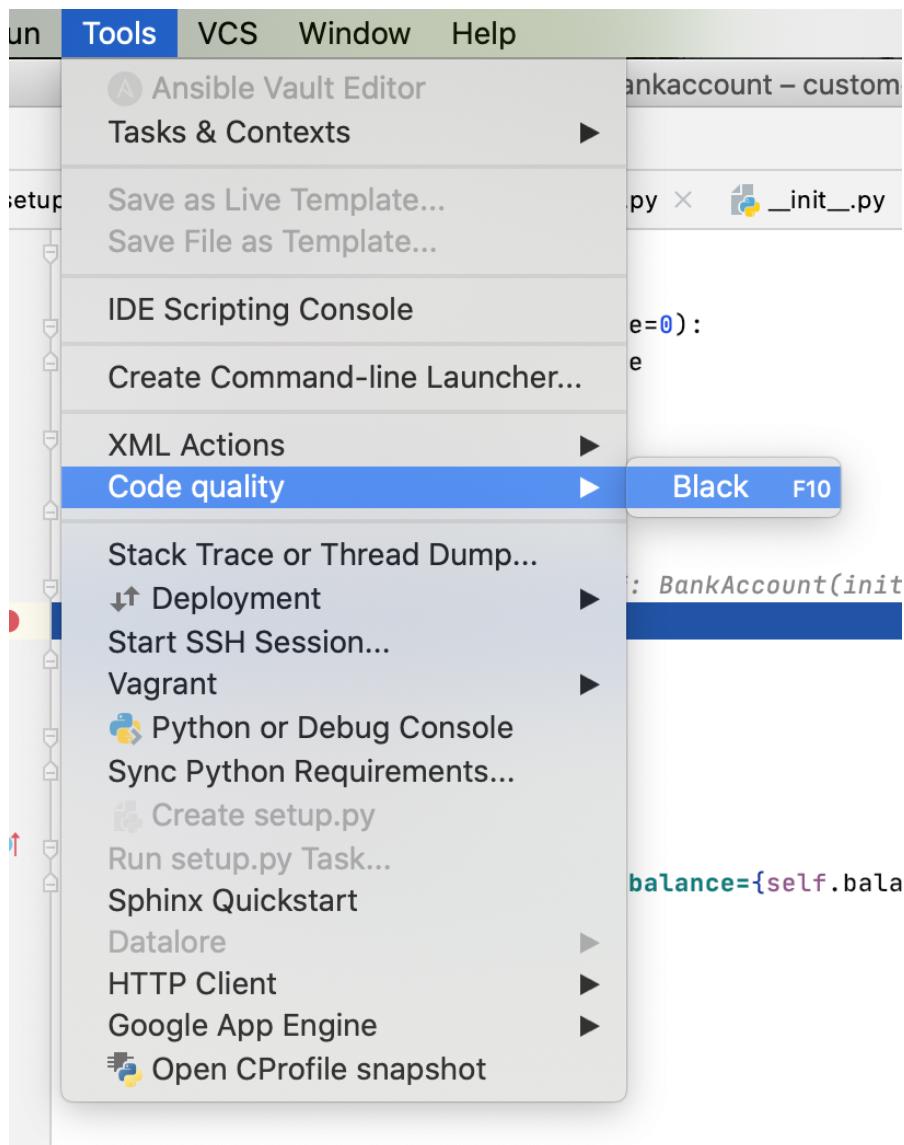
6.6 Adding additional tools to PyCharm

PyCharm is very extensible, and allows you to easily add additional tools to the application. First open the preferences and under the Tools menu select External Tools. Here select the + to add a new tool. This tool can be any executable callable on the PATH when PyCharm is open.



1. First you will be asked to give a Name, a Group (the menu item your tool will fall under) and a Description of the tool.
2. Then you will need to select the runnable executable, any additional parameters (the + will give you helpful templates), and the working directory.
3. In most cases the advanced options can be left alone, so select OK and you have extended PyCharm with your additional tool.

You can add custom keyboard shortcuts to run this tool but the easiest way to run this additional tool is from the Tools menu in the Group you defined when adding the tool to PyCharm.



You have now seen the basics of using PyCharm - the most powerful and customisable professional development IDE for Python. Of course it is well documented in the online help from JetBrains along with many blog posts and forums available online that will help you extend this IDE to meet your purposes. A good place to get started would be to read about version control

integration in PyCharm.

Chapter 7

Coding style

The preferred design philosophy for Python code is expressed flippantly as the “Zen of Python”:

import this

1

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to
do it.
Although that way may not be obvious at first unless you're
Dutch.

Now is better than never.

Although never is often better than *right* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good
↳ idea.

Namespaces are one honking great idea -- let's do more of
↳ those!

Code examples of these principles are given here:

http://artifex.org/~hblanks/talks/2011/pep20_by_example.html

The web development framework Django has its own design philosophies,
which many Python projects would do well to adhere to:

- Loose coupling
- Tight cohesion
- Less code (e.g. taking full advantage of the Python language)
- Quick development
- Don't repeat yourself
- Explicit is better than implicit
- Consistency at all levels
- Fat models, thin views

There are further design philosophies listed under various categories (e.g. models, URL design) here: <https://docs.djangoproject.com/en/dev/misc/design-philosophies/>.

7.1 Maintainability

Coding style is a matter of personal preference to some degree. For example, two attributes that are widely admired are **simplicity** and **generality**, but these may be in conflict: the simplest code may not be the most general.

An alternative is to strive for **Maintainability**.

An excellent essay on code maintainability and why it is such a useful goal is here: <http://www.advogato.org/article/258.html> (by Brad Cohen).

Code that is maintainable is easy to hack on to extend or modify. It is also easy to find and fix bugs in.

Maintainability Principle 1: Write less code!

A solid knowledge of Python’s language features is helpful for writing less code. Python offers many features useful for eliminating redundant code and boilerplate and thinking in higher-level abstractions:

- decorators
- context managers
- list comprehensions
- generator expressions

A thorough knowledge of Python’s builtins and standard library is also useful in avoiding reinventing the wheel. For example:

- the built-in functions `enumerate` and `zip`
- `namedtuple`, `Counter`, and `OrderedDict` in the `collections` module
- the `itertools` module

Maintainability Principle 2: Write test code!

Tests should be written against the interface, not the implementation. The best way to ensure this is to write the tests first. This is Test Driven Development (TDD). All the research points to this as yielding better code. It can also make coding and debugging easier if done consistently.

Maintainability Principle 3: Many small functions

A “functional” style of programming goes hand-in-hand with the above two principles. Code that uses many small functions without “side-effects” tend to be shorter, more composable, and easier to test.

Although Python is not a purely functional language, there are useful 3rd-party functional modules that provide functional features inspired by languages such as Scala, Clojure, and Haskell. One is **toolz**:

<http://toolz.readthedocs.org>

This is just a `pip install` away. Toolz is covered in another chapter / session.

7.2 PEP8

When in Rome, do as the Romans do.

— Roman proverb

Python has an official style known as PEP8, or Python Enhancement Proposal number 8. It provides specific guidelines that essentially all code should adhere to. All Python programmers should read it at least once. Most important:

- An indentation level should consist of exactly 4 space characters
- Lines should not be longer than 80 characters
- Function, variable, and method names should be lowercase with underscores: `my_variable`, not `myVariable` or `MyVariable`.
- Class names should use a leading capital and camelCase: `class DeckOfCards`.
- Avoid builtin names.
- **Ignore PEP8, when it makes sense to do so.**

The overarching principle is: *READABILITY COUNTS*.

```
dir(__builtin__)
```

1

```
['ArithmeticError',
 'AssertionError',
 'AttributeError',
 'BaseException',
 'BlockingIOError',
 'BrokenPipeError',
 'BufferError',
 'BytesWarning',
 'ChildProcessError',
 'ConnectionAbortedError',
 'ConnectionError',
 'ConnectionRefusedError',
 'ConnectionResetError',
 'DeprecationWarning',
 'EOFError',
 'Ellipsis',
 'EnvironmentError',
```

```
'Exception',
'False',
'FileExistsError',
'FileNotFoundException',
'FloatingPointError',
'FutureWarning',
'GeneratorExit',
'IOError',
'ImportError',
'ImportWarning',
'IndentationError',
'IndexError',
'InterruptedError',
'IsADirectoryError',
'KeyError',
'KeyboardInterrupt',
'LookupError',
'MemoryError',
'ModuleNotFoundError',
'NameError',
'None',
'NotADirectoryError',
'NotImplemented',
'NotImplementedError',
' OSError',
'OverflowError',
'PendingDeprecationWarning',
'PermissionError',
'ProcessLookupError',
'RecursionError',
'ReferenceError',
'ResourceWarning',
'RuntimeError',
'RuntimeWarning',
'StopAsyncIteration',
'StopIteration',
'SyntaxError',
'SyntaxWarning',
'SystemError',
```

```
'SystemExit',
'TabError',
'TimeoutError',
'True',
'TypeError',
'UnboundLocalError',
'UnicodeDecodeError',
'UnicodeEncodeError',
'UnicodeError',
'UnicodeTranslateError',
'UnicodeWarning',
'UserWarning',
'ValueError',
'Warning',
'ZeroDivisionError',
'__IPYTHON__',
'__build_class__',
'__debug__',
'__doc__',
'__import__',
'__loader__',
'__name__',
'__package__',
'__spec__',
'abs',
'all',
'any',
'ascii',
'bin',
'bool',
'bytearray',
'bytes',
'callable',
'chr',
'classmethod',
'compile',
'complex',
'copyright',
'credits',
```

```
'delattr',
'dict',
'dir',
'display',
'divmod',
'enumerate',
'eval',
'exec',
'filter',
'float',
'format',
'frozenset',
'get_ipython',
'getattr',
'globals',
'hasattr',
'hash',
'help',
'hex',
'id',
'input',
'int',
'isinstance',
'issubclass',
'iter',
'len',
'license',
'list',
'locals',
'map',
'max',
'memoryview',
'min',
'next',
'object',
'oct',
'open',
'ord',
'pow',
```

```
'print',
'property',
'range',
'repr',
'reversed',
'round',
'set',
'setattr',
'slice',
'sorted',
'staticmethod',
'str',
'sum',
'super',
'tuple',
'type',
'vars',
'zip']
```

Exercise

Copy this code to a new cell and then fix it to adhere to PEP8:

```
class bank_account:
    def __init__(self, name, balance):
        self.name = name # the name of the account holder
        self.balance = balance # the account balance

    def getBalance(self):
        return balance

    def deposit (self, amount):
        self.balance = self.getBalance() + amount

    def withdraw(self, amount):
        self.balance -= amount
        return amount
```

2

Now, copy the original code again and paste it into a Spyder or PyCharm editor window. Look for the hints provided by these editors about PEP8 violations.

7.3 PEP257 and the NumPy docstring standard

We've also seen *docstrings*, special strings that, placed underneath a function, become attached to that function and provide built-in documentation. Python also has a docstring style standard, PEP257, that you should also read. The main guidelines are:

- Docstrings are delimited by triple double-quote marks:

```
def function():
    """Docstring"""
    pass
```

- Docstrings should be written in the present imperative: "Do X", not "Does X".

```
def mass_of_universe():
    """Make complex calculations, return the total mass of the universe"""
    pass
```

- One line docstrings should have the opening and closing quotes on the same line (as above). Multi-line docstrings should have the first line as a one-liner, followed by a blank line, and then a longer description.

```
def mass_of_universe():
    """Make complex calculations, return the total mass of the universe"""

    The mass of the Universe depends on the size of the observable universe, which can be estimated from the distance to the farthest known supernovae, and on the density of matter in space, which can be measured by measurements of the cosmic microwave background.
```

This function commandeers several state-of-the-art telescopes to make the above measurements and then computes the answer. It makes use of the `astropy` library to handle units and conversions between different coordinate systems.

```
    return – now might be a good time to get a coffee.  
    """  
  
    ...
```

For technical / numerical / analytical code, PEP257 may be too broad and not explicit enough. In this case an alternative standard is the NumPy docstring standard. This is described in detail here. Projects like NumPy, SciPy, Matplotlib and Pandas use the NumPy docstring format. Here is an example:

```
1  
class Series:  
    def plot(self, kind, color='blue', **kwargs):  
        """  
        Generate a plot.  
  
        Render the data in the Series as a matplotlib plot of  
        the  
        specified kind.  
  
        Parameters  
        -----  
        kind : str  
            Kind of matplotlib plot.  
        color : str, default 'blue'  
            Color name or rgb code.  
        **kwargs  
            These parameters will be passed to the matplotlib  
        plotting  
            function.  
        """  
    pass
```

Extended exercise: coding style (PyLint / PEP8)

Write a script to detect duplicate files based on the md5 checksum of their contents. Read the PEP 8 style guidelines and make your code conform to the guidelines – especially whitespace, variable naming, capitalization, and imports.

Hint 1: use the `md5` class in the `hashlib` module and its `hexdigest` method. Example:

```
import hashlib  
hashlib.md5(b'file contents as a byte-string').hexdigest()
```

5

```
'011f74e20f6d18317e1d1b0044a279e0'
```

Hint 2: use the `defaultdict` class in the `collections` module or (easier) the `groupby` function in the 3rd-party `toolz` package to map md5 hashes to lists of filenames.

```
# Example solution: note the scan() function in particular:  
# See solutions/dup_detector.py
```

1

Chapter 8

Linting

Linting is the process of running a program that will analyze code for potential errors and issues.

The term was coined by Stephen C. Johnson of Bell Labs in 1978 in relation to a grammar he was writing for C code. Since then linting tools (“linters”) have proliferated across many programming languages, including Python.

In compiled languages such as C++ or Rust you will have a compilation step that will uncover many issues and will warn you of illegal constructs and situations that, while allowed, are likely to be problematic.

If your script is not syntactically valid Python will give you a `SyntaxError` that will tell you where the syntax that could not be parsed was found.

For example in this code we are missing a semicolon after the `if` statement:

```
>>> if 10 == 20
      File "<stdin>", line 1
          if 10 == 20
              ^
SyntaxError: invalid syntax
```

1

Beyond this because Python is an interpreted language so you need to use external utilities if you wish to get information about warnings and potential

run time errors ahead of actually running the code.

The Python Code Quality Authority (PyCQA)

PyCQA, by their own admission,

is not actually an authority on anything

Instead, they are a loose collection of groups working on similar tools that are used for linting and code analysis. These are at this time:

- Astroid
- Baron
- Flake8
- mccabe
- pycodestyle
- pep8-naming
- pydocstyle
- Pylint
- RedBaron

You will be looking at two examples - the lighter weight `pycodestyle` and the more powerful (but more complex) `pylint`.

8.1 `pycodestyle`

In Python the *de facto* standard is to follow the Python Enhancement Proposal 8 (PEP8) style guide when writing code. In the most basic case you can analyze a piece of code and whether or not it conforms to PEP8 and significantly improve the quality of the code.

The `pycodestyle` package (which used to be known as `pep8`) is entirely devoted to checking the style of code against the PEP8 specification. It is simple to install with `pip`:

```
pip install pycodestyle
```

And very simple to run:

```
pycodestyle some_script.py
```

Consider the following code. This is valid Python, runs and executes exactly as you would expect. However it is not particularly readable, and violates the PEP8 guidelines in several places.

```
%%writefile solutions/ugly_script.py
import os, sys, string
def S(s):

    N = ''
    for I in range(len(s)):
        if s[I] not in string.ascii_uppercase:
            N = N + s[I]
    return N
print(S('I think, therefore I am a Weasel'))
```

Overwriting ugly_script.py

Running the above script in Python with:

```
python ugly_script.py
```

Produces the output we expect, namely:

```
" think, therefore am a easel"
```

But the code is ugly, inefficient, and hard to understand the purpose. If we check this code with pycodestyle:

```
pycodestyle ugly_script.py
```

We get the following list of issues:

```
! pycodestyle ugly_script.py
```

```
ugly_script.py:1:10: E401 multiple imports on one line
ugly_script.py:2:1: E302 expected 2 blank lines, found 0
ugly_script.py:3:1: W293 blank line contains whitespace
ugly_script.py:4:1: W293 blank line contains whitespace
ugly_script.py:5:5: E303 too many blank lines (2)
```

```
ugly_script.py:10:1: E305 expected 2 blank lines after class
  ↵ or function definition, found 0
ugly_script.py:10:45: W292 no newline at end of file
```

Each line of the output represents an issue with the code found by this linting tool (i.e. a code that violates a PEP8 guideline). Each line can be broken up as follows:

```
{file_name}:{line_number}:{column_number}: {error_code} {error_...}
```

The `error_code` is 4 characters long and, and is either an `E` or a `W` followed by 3 numbers identifying the Error or Warning respectively. The `error_message` is everything else in the line.

For example, the first message is `ugly_script.py:1:10: E401 multiple imports on one line`, which can be broken up as:

Parameter	Value
file_name	ugly_script.py
line_number	1
column_number	10
error_code	E401
error_message	multiple imports on one line

Fixing the code according to these errors gives the following:

```
%%writefile solutions/less_ugly_script.py
import os
import sys
import string

def S(s):
    N = ''
    for I in range(len(s)):
        if s[I] not in string.ascii_uppercase:
            N = N + s[I]
    return N
```

4

```
print(S('I think, therefore I am a Weasel'))
```

Overwriting `less_ugly_script.py`

While there are still substantial changes that could be made to this script to improve code, even simple changes to the arrangement of lines of code in the file to follow the PEP8 convention substantially improves readability in the file.

8.2 `pylint`

`pylint` is a much more substantial code quality checker. It works by doing a static analysis run through of your code. This means it analyses your code for issues without running it.

`pylint` checks not only for PEP8 style violations, but for a variety of errors and potential problems. Along with reporting the issues with code quality, `pylint` will give you a score out of 10 for the quality of your code (and negative scores are possible). This can be useful as a measure to set yourself when writing code, as `pylint` will report between checks the amount this score has changed.

`pylint` is once again easy to install via pip:

```
pip install pylint
```

And you can check a file using `pylint` with:

```
pylint some_script.py
```

Using the above script to check quality with `pylint` you get a much longer output:

```
! pylint ugly_script.py
```

```
No config file found, using default configuration
***** Module ugly_script
```

```
C: 3, 0: Trailing whitespace (trailing-whitespace)
C: 4, 0: Trailing whitespace (trailing-whitespace)
C: 10, 0: Final newline missing (missing-final-newline)
C: 1, 0: Missing module docstring (missing-docstring)
C: 1, 0: Multiple imports on one line (os, sys, string)
    ↵ (multiple-imports)
C: 2, 0: Function name "S" doesn't conform to snake_case
    ↵ naming style (invalid-name)
C: 2, 0: Argument name "s" doesn't conform to snake_case
    ↵ naming style (invalid-name)
C: 2, 0: Missing function docstring (missing-docstring)
C: 5, 4: Variable name "N" doesn't conform to snake_case
    ↵ naming style (invalid-name)
C: 6, 4: Consider using enumerate instead of iterating with
    ↵ range and len (consider-using-enumerate)
C: 6, 8: Variable name "I" doesn't conform to snake_case
    ↵ naming style (invalid-name)
C: 8,12: Variable name "N" doesn't conform to snake_case
    ↵ naming style (invalid-name)
W: 1, 0: Unused import os (unused-import)
W: 1, 0: Unused import sys (unused-import)
```

```
-----]
↳
Your code has been rated at -7.50/10 (previous run: -7.50/10,
    ↵ +0.00)
```

The format of each issue is similar (though not quite identical) to pycodestyle:

```
{error_category}: {line_number}, {column_number}: {error_message}
```

Again

Fixing the above issues will take longer, but will result in a much nicer script:

```
%%writefile solutions/nicer_script.py
```

```
"""
```

```
Much nicer script.
```

```
All thanks to pylint.
```

```
"""
import string

def remove_uppercase(candidate):
    """
    All functions should have docstrings.

    In this case, remove capital letters from some candidate
    string. Return a new string with the letters removed.
    """
    shortened = ''
    for letter in candidate:
        if letter not in string.ascii_uppercase:
            shortened += letter
    return shortened

print(remove_uppercase('I think, therefore I am a Weasel'))
```

Overwriting `nicer_script.py`

8.3 Warnings vs errors

When Pylint reports an error it is something that is very likely to make your program break. A warning is something that may be valid but is often going to be problematic.

Let's look at an example of a Warning with an unused variable and how this differs from an Error with an undefined variable.

```
def unused_variable():
    """
    We define two variables here but only ever use one of
    ↵ them"""
    unused = 5
    used = 10
    return used * 2
```

1

In this case we have a valid function that won't break at run time but has a variable called unused that never gets used. A good coding style is to never have any unused variables present in the source code. They tend to cause problems in the future. Because this is a situation that leads to future problems Pylint will warn us about this function:

```
pylint_examples (master)$ pylint unused_variable.py
*****
Module unused_variable
unused_variable.py:5:4: W0612: Unused variable 'unused'
    ↵ (unused-variable)

-----
Your code has been rated at 7.50/10
```

```
def undefined_variable():
    """This function contains an error where there's a
    ↵ variable 'X' instead of 'x'"""
    X = 5
    z = x * 2 # NameError: name 'x' is not defined

    return z
```

In the above code we have an undefined variable that we are attempting to use, this will fail with a `NameError: name 'x' is not defined` when we try to execute the function `undefined_variable()`. Pylint will report that this is an error.

Here's what this looks like when we scan it with Pylint:

```
pylint_examples (master)$ pylint undefined_variable.py
*****
Module undefined_variable
undefined_variable.py:5:4: C0103: Variable name "Number"
    ↵ doesn't conform to snake_case naming style (invalid-name)
undefined_variable.py:6:11: E0602: Undefined variable
    ↵ 'number' (undefined-variable)
undefined_variable.py:5:4: W0612: Unused variable 'Number'
    ↵ (unused-variable)
```

```
-----  
↳  
Your code has been rated at -13.33/10 (previous run:  
↳  -13.33/10, +0.00)
```

8.4 Take Pylint errors seriously

When Pylint reports an `Error` it almost always means the code has a serious defect, make sure all errors Pylint discovers are fixed. If you run into a bug it is a good idea to run through the code with Pylint, sometimes it will uncover the reason or give you clues as to the right places to look further.

You can check your code for errors only with the `pylint -E` invocation, this is a good command to put into your Continuous Integration pipeline to check for errors.

8.5 Take warnings seriously too!

Pylint warnings are often indicative of serious issues. If you have a situation where pylint is reporting a false-positive you may wish to disable the warning for that line, this will show the people you are working with that you have addressed the issue.

Chapter 9

Writing elegant Python

```
%run preamble.py
```

1

In any programming language you can write incomprehensible code. This is not a good thing - whether it is possible to understand how your code works, or even to update it is dependent on the way you write your code. Happily working in Python it is easy to write elegant code.

Writing elegant code means two things:

1. Writing the minimum amount of code possible to achieve your goal in the most efficient way possible.
2. Writing code that conforms to a shared standard and understanding with meaningful variable and classnames, that is well documented, and intended to be usable long into the future.

This section includes practical examples of updating common code patterns to be as efficient and elegant as possible.

```
# Load data:  
apple = pd.read_hdf('Data/AAPL.h5')  
forbes = pd.read_csv('Data/forbes1964.csv').set_index('name')
```

9

9.1 Iterate through sequences, not indices

Python iterators return element at a time from any sequence being iterated over. Having to create a sequence of indexes just to extract values from a sequence is a wasted step.

```
# Not:  
for i in range(len(apple.columns)):  
    col_name = apple.columns[i]  
    print(col_name, end=' | ')
```

10

Open|High|Low|Close|Volume|Adj Close|

```
# Instead:  
for col_name in apple.columns:  
    print(col_name, end=' | ')
```

11

Open|High|Low|Close|Volume|Adj Close|

9.2 Use `zip()` to iterate through sequences in lockstep

Often a reason for creating sequences containing nothing but indexes is to extract data from more than one collection at once. In Python this is not needed - instead in Python you can use `zip` to extract values from sequences in lockstep.

```
countries = ['Australia', 'Netherlands', 'China',  
            'Mozambique']  
capitals = ['Canberra', 'Amsterdam', 'Beijing', 'Maputo']  
populations = [24598933, 17132854, 1386395000, 29668834] #  
            ← 2017 population, source: World Bank
```

12

```
for country, capital, pop in zip(countries, capitals,  
    ↵ populations):  
    print(country, capital, pop, sep='|')
```

13

```
Australia|Canberra|24598933  
Netherlands|Amsterdam|17132854  
China|Beijing|1386395000  
Mozambique|Maputo|29668834
```

Especially useful: use `zip()` to construct dictionaries from two sequences

```
dict(zip(countries, populations))
```

14

```
{'Australia': 24598933,  
 'Netherlands': 17132854,  
 'China': 1386395000,  
 'Mozambique': 29668834}
```

Or to invert a dictionary

```
colour_codes = {'r': 'red', 'b': 'blue', 'k': 'black', 'y':  
    ↵ 'yellow'}
```

15

```
# To invert it:  
colour_codes_inv = dict(zip(colour_codes.values(),  
    ↵ colour_codes.keys()))
```

16

Check that there are no duplicate keys, though: these would be overwritten.

```
assert len(colour_codes) == len(colour_codes_inv)
```

17

```
airports = pd.read_csv('./Data/airports.csv')
```

18

```
airports[:3]
```

19

	Airport_ID	Name	City	Country
0	1	Goroka	Goroka	Papua New Guinea
1	2	Madang	Madang	Papua New Guinea
2	3	Mount Hagen	Mount Hagen	Papua New Guinea
IATA_FAAC	ICAO	Latitude	Longitude	
GKA	AYGA	-6.081689	145.391881	
MAG	AYMD	-5.207083	145.788700	
HGU	AYMH	-5.826789	144.295861	
Altitude	Timezone	DST	Tz_db_time_zone	
5282	10.0	U	Pa-	
20	10.0	U	cific/Port_Moresby	
5388	10.0	U	Pa-	
			cific/Port_Moresby	

```
# E.g. map the 3-letter code to the 4-letter ICAO code:
airport_codes = dict(zip(airports['IATA_FAAC'],
                           airports['ICAO']))
```

20

```
def get_size(file):
    return os.stat(file).st_size

files = glob.glob('/data/*')
sizes = [get_size(file) for file in files]
# More code here ...
```

21

If you later want a dictionary mapping filenames to their sizes, use:

```
file_sizes = dict(zip(files, sizes))
```

22

Check again for duplicate keys:

```
assert len(filesizes) == len(files)
```

23

9.3 Using the (extended) unpacking syntax

Python makes it very easy to unpack variables from collections, which is particularly useful when getting multiple values back from functions:

```
import requests

def get_longitude_latitude(city):
    url = 'https://llinvc1jc.execute-api.ap-southeast-1'
    ↵ 2.amazonaws.com/dev/location/query'
    params = {'name': city}
    response = requests.get(url, params)
    response.raise_for_status()
    location = response.json()
    return location['longitude'], location['latitude']
```

24

```
get_longitude_latitude('Batemans Bay')
```

25

```
(150.207306681, -35.6896187146)
```

```
x, y = get_longitude_latitude('Batemans Bay')
```

26

```
# Swap them:
(y, x) = (x, y)
```

27

This is also commonly encountered in loops, particularly when using `zip` (see above) or `enumerate`.

```
# In loops:  
for (i, col_name) in enumerate(forbes.columns):  
    print(i, col_name)
```

28

```
0 rank  
1 country  
2 sales  
3 profits  
4 assets  
5 marketvalue
```

Elegant unpacking of arrays

```
# Unpacking combined with transpose:  
profits, marketvalue = forbes[['profits',  
    ↵ 'marketvalue']].dropna().values.T
```

29

Limiting parameters

The extended unpacking syntax allows you to use * arguments to allow partial unpacking. The * argument can be used anywhere, but only once:

```
# Extended unpacking:  
slope, intercept, *_, = stats.linregress(profits,  
    ↵ marketvalue)
```

30

```
slope, intercept
```

31

```
(7.426327725367626, 8.971530346546103)
```

```
slope, *_, stderr = stats.linregress(profits, marketvalue)  
slope, stderr
```

32

```
(7.426327725367626, 0.2621443490142193)
```

```
*_, stderr = stats.linregress(profits, marketvalue)
```

33

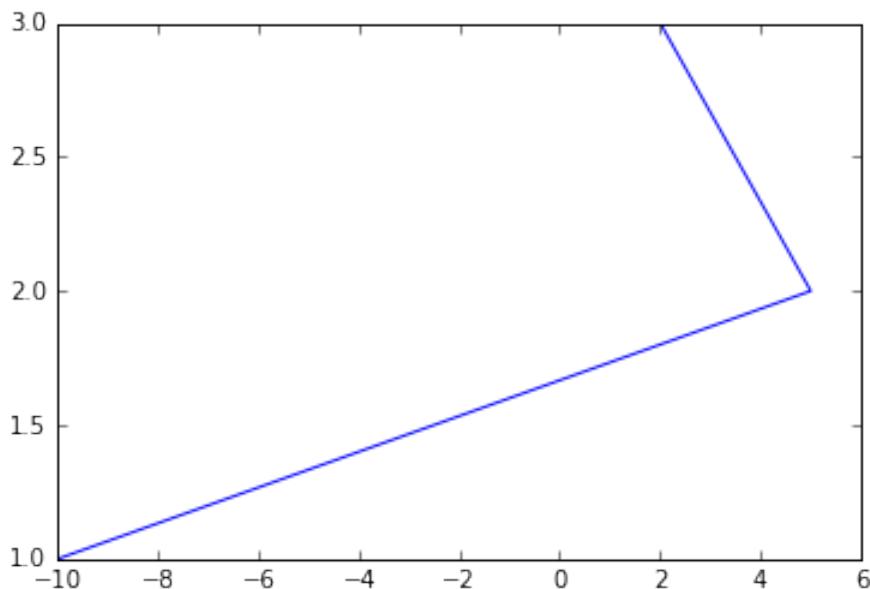
Exercise: write a wrapper function for `plt.plot(xs, ys, ...)` called `newplot` that takes a `transpose` keyword argument

```
def newplot(xs, ys, transpose=False):
    if transpose:
        ys, xs = xs, ys
    plt.plot(xs, ys)
```

34

```
newplot([1, 2, 3], [-10, 5, 2], transpose=True)
```

35



9.4 Argument expansion in function calls

Similar to argument unpacking, Python allows for argument expansion in function calls using `*` arguments to expand an iterable:

```
# This:  
print(*forbes.columns, sep=',')
```

36

rank, country, sales, profits, assets, marketvalue

```
# versus this:  
print(', '.join(forbes.columns))
```

37

rank, country, sales, profits, assets, marketvalue

Python also allows for the use of `**` arguments to expand dictionaries as keyword arguments:

```
params = {'sep': '|', 'end': '!'}  
print(*forbes.columns, **params)
```

38

rank|country|sales|profits|assets|marketvalue!

9.5 Use namedtuples (if immutable)

What does this code do?

```
p = (170, 0.1, 0.6)  
if p[1] >= 0.5:  
    print('That is bright!')  
if p[2] >= 0.5:  
    print('That is light!')
```

1

How about this code?

```
from collections import namedtuple  
  
Color = namedtuple('Color', ['hue', 'saturation',  
    ↴ 'luminosity'])  
  
p = Color(170, 0.1, 0.6)  
if p.saturation >= 0.5:  
    print('That is bright!')  
if p.luminosity >= 0.5:  
    print('That is light!')
```

1

This is the benefit of namedtuples: more readable code.

Example: linregress

An earlier example gave this:

```
params = stats.linregress(profits, marketvalue)
```

39

```
params
```

40

```
LinregressResult(slope=7.426327725367626,  
    ↴ intercept=8.971530346546103, rvalue=0.538988185427957,  
    ↴ pvalue=2.831606318704309e-148, stderr=0.2621443490142193)
```

This is another namedtuple. You can use this as follows:

```
params.slope, params.intercept, params.stderr
```

41

```
(7.426327725367626, 8.971530346546103, 0.2621443490142193)
```

which is more readable than this:

```
params[0], params[1], params[4]
```

42

```
(7.426327725367626, 8.971530346546103, 0.2621443490142193)
```

The namedtuple constructor is found in the inbuilt collections module:

```
from collections import namedtuple
```

43

A named tuple object is created with an object type name, which should be a valid Python identifier, and a list of field names. Once you have created the named tuple you can create an instance by passing either positional or keyword arguments. For example:

```
Currency = namedtuple('Currency', ['name', 'central_bank',  
    ↴ 'symbol'])
```

44

```
euro = Currency('Euro', 'European Central Bank', '€')  
euro
```

45

```
Currency(name='Euro', central_bank='European Central Bank',  
    ↴ symbol='€')
```

```
aus_dollar = Currency('Australian Dollar', 'Reserve Banks of  
    ↴ Australia', '$')  
aus_dollar
```

46

```
Currency(name='Australian Dollar', central_bank='Reserve Banks  
    ↴ of Australia', symbol='$')
```

```
aus_dollar.central_bank
```

47

```
'Reserve Banks of Australia'
```

Exercise: Building named tuples

1. Create your own namedtuple type to represent the following stats about a DataFrame: number of rows, number of columns, number of duplicated index values, total memory usage

2. Write a function that returns a new instance of your namedtuple for a given DataFrame

Chapter 10

Writing elegant Python – part 2

```
%run preamble.py
```

1

This chapter picks up from the earlier chapter “Writing elegant Python – part 1”.

10.1 Data Classes (if mutable)

Data Classes are a feature of the Python standard library since Python 3.7. (If you are using Python 3.6, they have also been backported and can be installed with `pip install dataclasses`.

Data Classes are a helper tool that will generate code to simplify building a Python class. They automatically generate the `__init__` method using Python’s recent type-hinting syntax. It is simple to make a dataclass simply by decorating a Python class:

```
from dataclasses import dataclass

@dataclass
class Currency:

    name: str
    central_bank: str
    symbol: str
    code: str = 'AUD'

    def get_exchange_rate(self, base='EUR'):
        url = 'http://data.fixer.io/api/latest'
        params = {
            'access_key':
'93136301b1c8a659c34b8ce6bb63d0fa',
            'symbols': self.code,
            'base': base
        }
        response = requests.get(url, params)
        response.raise_for_status()
        return response.json()['rates'][self.code]
```

```
euro = Currency('Euro', 'European Central Bank', '€', 'EUR')
euro
```

```
Currency(name='Euro', central_bank='European Central Bank',
         symbol='€', code='EUR')
```

```
aus_dollar = Currency('Australian Dollar', 'Reserve Banks of
                     Australia', '$')
aus_dollar
```

```
Currency(name='Australian Dollar', central_bank='Reserve Banks
         of Australia', symbol='$', code='AUD')
```

```
aus_dollar.get_exchange_rate(base=euro.code)
```

53

1.579552

The biggest advantage of a dataclass is that you don't have to spend time implementing the stock standard `__init__`, `__eq__`, `__repr__`, and other methods. This can potentially lead to shorter, more readable class definitions.

One disadvantage to data classes is that they add their own complexity from auto-generated (hidden) code; it can be harder to fathom exactly how a class is defined.

Another drawback is that they impose the requirement that parameter names for the `__init__` method be identical to attribute names. If you wish to use the common (PEP8-blessed) notation of a leading underscore for weakly indicating an internal attribute, data classes then force you also to require the underscores in the parameter names to `__init__`, which leads to uglier APIs:

```
@dataclass
class Card:
    _rank: str
    _suit: str

# This fails:
ace_of_spades = Card(rank='A', suit='spades')          #
← TypeError

# Instead:
ace_of_spades = Card(_rank='A', _suit='spades')        # uglier
```

1

Exercise: Updating the Currency dataclass

1. Make the `central_bank` optional (default value `None`).
2. Modify the dataclass above for currency to include an `amount` attribute which should be a `float`, defaulting to `1.0`.

3. Add a new method `convert_to(self, target_currency: Currency)` to the class that returns a new `Currency` object in the target currency with the correct value in the `amount` attribute.

For example: if you have SGD 100, how much is this worth in USD?

10.2 Keep exception blocks small

Keep exception-handling blocks as small as possible, and catch specific exceptions.

Exceptions in Python are not scary. You should think of them instead as extremely useful tools in finding out what has gone wrong in incoming data that you may have no control over. This is in contrast to using exception handling to gloss over issues with your code. Rather than using exception handling here you should be writing unit tests to test and fix your code! Part of using exceptions to your advantage is to write small, explicit exception handling blocks in your code rather than large broad exceptions which don't tell you what's wrong.

Consider this scenario: Every day you run a process where you read from an internal database containing a list of stock options to watch. Taking this information you must connect to an external API and retrieve the closing prices of these stocks over the last 12 months, along with the current price. You could write your code like this:

```
# What not to do
try:
    def stocks_of_interest():
        """
        Get the list of stocks from the database
        """
        #...
        pass

    def query_stock_prices(stock_symbol):
        """
        Get stock prices from external API
        """
```

54

```
"""
pass

stocks = stocks_of_interest()
for stock in stocks:
    data = query_stock_prices(stock)

# and so on
except:
    pass
```

This code silently fails with no explanation why. There are two obvious sources of failure:

1. The connection to the database fails and for whatever reason doesn't return any data. This is a failure within your system somewhere.
2. The query of stock prices fails when you query the external API. This is a failure of a service you're paying money to be part of.

With the code written like this you don't know where the failure is. Rewriting the code to catch exceptions on discrete pieces of work gives you a much better idea of what's gone wrong:

```
def stocks_of_interest():
    """
    Get the list of stocks from the database
    """
    #...
    return ['AAPL', 'MSFT']

def query_stock_prices(stock_symbol):
    """
    Get stock prices from external API
    """
    raise ValueError("Could not connect to server")

try:
    stocks = stocks_of_interest()
```

55

```
except IOError as e:  
    print(str(e))  
  
for stock in stocks:  
    try:  
        data = query_stock_prices(stock)  
    except ValueError as e:  
        print(f"Could not retrieve value for stock {stock} -  
        {e}")
```

Could not retrieve value for stock AAPL - Could not connect to
↳ server

Could not retrieve value for stock MSFT - Could not connect to
↳ server

You should also consider that you can use exception handling to handle different types of errors explicitly. For example you could have separate errors for failing to connect to the external stock price service at all vs not being able to find a given stock price symbol, and handle them differently:

```
class NoServerConnection(Exception):  
    pass  
class NoStockFound(Exception):  
    pass  
  
def query_stock_prices(stock_symbol):  
    """  
    Get stock prices from external API  
    """  
    if stock_symbol == 'AAPL':  
        raise NoStockFound('Could not find stock in service')  
    raise NoServerConnection("Could not connect to server")  
  
for stock in stocks:  
    try:  
        data = query_stock_prices(stock)  
    except NoStockFound as e:
```

56

```

    print(f"Could not retrieve value for stock {stock} - "
        ↵  {e}")
except NoServerConnection as e:
    print(f'Could not connect to server')
    raise # deliberately throw an error if you can't
        ↵  connect to the external server

```

Could not retrieve value for stock AAPL - Could not find stock
 ↵ in service
 Could not connect to server

Exercise: debug this and rewrite the exception handling:

```

def get_hash(file):
    """
    Returns the MD5 hex digest for a file's contents
    """
    try:
        with open(file, mode='rb') as f:
            data = f.read()
        return hashlib.md5(data).hex_digest()
    except:
        return None

result = get_hash('/data/AAPL.h5')
print(result)

```

None

10.3 Define `__iter__` on classes as a generator function

When you are writing your own class objects in Python you may be familiar with the `__next__` method which, when implemented in a class,

allows you to define a loop over that class and return some values, raising a `StopIteration` exception when the list is exhausted. Consider the simple example of a counter:

```
@dataclass
class SimpleFibonacci:

    stop: int

    current = 1
    increment = 0

    def __iter__(self):
        return self

    def __next__(self):
        current = self.current
        self.current += self.increment
        self.increment = current

        if current > self.stop:
            raise StopIteration
        return current
```

```
list(SimpleFibonacci(20))
```

```
[1, 1, 2, 3, 5, 8, 13]
```

However, if you don't need to maintain the state of the iterator you can yield from the `__iter__` method instead. When there are no more yield statements the `StopIteration` exception will be raised automatically. The most basic example is:

```
class YieldZeroOne:

    def __iter__(self):
```

```
yield 0
yield 1
```

```
list(YieldZeroOne())
```

101

[0, 1]

Exercise: Implement `__iter__` to return only active services

Given this code example write an `__iter__` method that loops over `self.services` and yields the service only if it is active.

```
# Example: define a class for a Server that allows iteration
# only through active services:
class Server:
    """Taken from https://opensource.com/article/18/4/
    elegant-solutions-everyday-python-problems"""

    services = [
        {'active': False, 'protocol': 'ftp', 'port': 21},
        {'active': True, 'protocol': 'ssh', 'port': 22},
        {'active': True, 'protocol': 'http', 'port': 21},
    ]

    def __iter__(self):
        # Implement solution here
        pass
```

99

Test your solution with:

```
server = Server()
for service in server:
    print(service['protocol'], service['port'])
```

98

```
ssh 22  
http 21
```

10.4 Use `islice` to extract the first `n` items from a generator

Assume that you've written a generator that will keep generating new values indefinitely. Modifying the Fibonacci example from earlier you could have:

```
class FibonacciInfinite:  
  
    def __iter__(self):  
        current = 1  
        increment = 0  
        while True:  
            yield current  
            current, increment = current + increment, current
```

```
for i, f in enumerate(FibonacciInfinite()):  
    if i >= 5:  
        break  
    print(f)
```

```
1  
1  
2  
3  
5
```

This can be a bit of a pain as you have to write loops or list comprehensions to capture these values. Instead, you can use the inbuilt `itertools` module and the `islice` function to extract values from the iterator:

```
from itertools import islice
```

```
islice(FibonacciInfinite(), 10)
```

110

```
<itertools.islice at 0x1190b9f48>
```

This returns a generator of its own (in fact you are wrapping your infinite generator in another generator), but you can easily cast this as a `list` to extract the values:

```
list(islice(FibonacciInfinite(), 10))
```

109

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

As a bonus, you can add optionally use the start, stop, step syntax similar to the `range` function:

```
list(islice(FibonacciInfinite(), 0, 10, 2))
```

111

```
[1, 2, 5, 13, 34]
```

Note like `range` the step defaults to 1, but unlike `range` the step must always be a positive integer.

10.5 Use `itertools` for getting combinations of collections

The `itertools` module is actually extremely useful if you are dealing with many different iterable objects or collections of data. Consider you have some set of people you are trying to put into groups:

```
people = ['Henry', 'Hannah', 'Marcie']
```

114

You know the formula for permutations: $n P_k = \frac{n!}{(n-k)!}$, and you could write a series of nested for loops to figure out groups of 2 like so (expecting 6 possible permutations with 3 people, given order is important):

```
for a in people:  
    for b in (p for p in people if a != p):  
        print(a, b)
```

115

```
Henry Hannah  
Henry Marcie  
Hannah Henry  
Hannah Marcie  
Marcie Henry  
Marcie Hannah
```

Of course as the collection of people gets larger, and the number of people in the group get larger this becomes a deep nested for loop, and difficult to maintain. Out of the box itertools comes with a permutations function:

```
from itertools import permutations
```

116

```
for a, b in permutations(people, 2):  
    print(a, b)
```

117

```
Henry Hannah  
Henry Marcie  
Hannah Henry  
Hannah Marcie  
Marcie Henry  
Marcie Hannah
```

If you wish to calculate permutations with replacement you can use the product method. This method is a shortcut to writing nested for loops:

```
from itertools import product
```

118

```
for a, b in product(people, people):  
    print(a, b)
```

119

```
Henry Henry
Henry Hannah
Henry Marcie
Hannah Henry
Hannah Hannah
Hannah Marcie
Marcie Henry
Marcie Hannah
Marcie Marcie
```

It also has a shortcut for repeating the same iterable object a number of times, yielding n^k results:

```
for a, b in product(people, repeat=2):
    print(a, b)
```

127

```
Henry Henry
Henry Hannah
Henry Marcie
Hannah Henry
Hannah Hannah
Hannah Marcie
Marcie Henry
Marcie Hannah
Marcie Marcie
```

Likewise itertools comes with a `combinations` (where order is not important) method that returns $nC_k = \frac{n!}{k!(n-k)!}$ results, and a `combinations_with_replacement` method that returns $({}^n)_k = {}^{n+k-1}C_k = \frac{(n+k-1)!}{k!(n-1)!}$ results:

```
from itertools import combinations,
    combinations_with_replacement
```

118

```
# expecting 3 results for combinations of size 2
for a, b in combinations(people, 2):
    print(a, b)
```

119

```
Henry Hannah
Henry Marcie
Hannah Marcie
```

```
# expecting 6 results for combinations with replacement with
    ↵ a group size of 2
for a in combinations_with_replacement(people, 2):
    print(*a)
```

123

```
Henry Henry
Henry Hannah
Henry Marcie
Hannah Hannah
Hannah Marcie
Marcie Marcie
```

Exercise: Calculate the possible combinations for a bike lock

You have a bike lock where you can select the values 0 — 9 on each of the 4 wheels. Use the appropriate method from itertools to calculate all the possible combinations. What would be the effect of increasing the possible values on each wheel to 16?

```
# %load solutions/bike_lock.py
```

6

10.6 Memoize the results of long running functions

Often a function will be completely deterministic based on the input, but takes a long time to run. For example, the following function returns the location of the closest city to a given latitude and longitude:

```
import requests  
import time
```

6

```
def get_closest_city(latitude, longitude):  
    time.sleep(5)  
    url = 'https://llinvfuljc.execute-api.ap-southeast-'  
    ↵ 2.amazonaws.com/dev/location/nearest'  
    params = {'latitude': latitude, 'longitude': longitude}  
    response = requests.get(url, params)  
    response.raise_for_status()  
    return response.json()['name']  
  
get_closest_city(52.4, 4.9)
```

7

'Amsterdam'

Given the location of the server, the length of time the server takes to execute commands (in this case we sleep for 5 seconds), and potential limitations on usage of the API, if you are going to keep calling this function it can have a dramatic effect on execution time.

Fortunately the `functools` library in Python contains a method for a `lru_cache` (LRU stands for Least Recently Used, which is the item that will be dropped from the cache when the cache size is exceeded). Using this as a Python decorator we can decorate the function call like so:

```
from functools import lru_cache
```

8

```
@lru_cache()  
def get_closest_city(latitude, longitude):  
    time.sleep(5)  
    url = 'https://llinvfuljc.execute-api.ap-southeast-'  
    ↵ 2.amazonaws.com/dev/location/nearest'  
    params = {'latitude': latitude, 'longitude': longitude}  
    response = requests.get(url, params)
```

10

```
response.raise_for_status()
return response.json()['name']

get_closest_city(52.4, 4.9)
```

```
'Amsterdam'
```

The first time the call to the function will take the full amount of time. Subsequent calls to this function with the same arguments however will be much faster, simply retrieving the result from the cache:

```
%%timeit
get_closest_city(52.4, 4.9)
```

12

```
171 ns ± 2.35 ns per loop (mean ± std. dev. of 7 runs,
↪ 10000000 loops each)
```

10.7 Implementing a persistent cache

It may be useful to implement a persistent cache in Python that stores results between sessions. The use case here is similar to memoization but has the extra step of saving the results to a persistent storage like disk involved as well. However to make this more generic we need to see some more complex processes:

First, if you want to decorate a function such that the results can be stored persistently for some given input arguments you need to be able to programmatically get the function arguments, which can be done via the inbuilt `inspect` module:

```
def get_closest_city(latitude, longitude):
    time.sleep(5)
    url = 'https://llinvcf1jc.execute-api.ap-southeast-'
    ↪ 2.amazonaws.com/dev/location/nearest'
    params = {'latitude': latitude, 'longitude': longitude}
    response = requests.get(url, params)
```

14

```
response.raise_for_status()  
return response.json()['name']
```

```
import inspect
```

20

```
sig = inspect.signature(get_closest_city)  
sig.parameters.keys()
```

23

For ease of use you'll want to store strings of the incoming values. This does mean that for custom classes you should implement an appropriate string method. You'll use the `pickle` library to store the objects directly.

```
import pickle
```

40

```
pickle.dumps(43.5)
```

46

```
b'\x80\x03G@E\xc0\x00\x00\x00\x00\x00\x00.'
```

```
pickle.loads(pickle.dumps(43.5))
```

47

43.5

To store the data you'll use the inbuilt `sqlite3` module to create a sqlite (on-disk) database for storage. In this case you'll use the `wraps` method in `functools` to build your decorator function:

```
import sqlite3  
from functools import wraps  
from itertools import chain
```

57

```
def persistent_cache_wrapper(database_location=':memory:'):
    def decorator(f):
        """Create the storage cache and return the decorated
        ↵ function"""

        sig = inspect.signature(get_closest_city)
        column_defs = ', '.join(
            f'{k} blob' for k in sig.parameters.keys()
        )
        conn = sqlite3.connect(database_location)
        create_table = f"""
            create table if not exists {f.__name__}
        ↵ ({column_defs}, _result blob)
        """
        c = conn.cursor()
        c.execute(create_table)
        conn.commit()

    @wraps(f)
    def wrapper(*args, **kwargs):
        """Firstly query the decorated cache location"""
        ordered_incoming_args = chain(
            args,
            (kwargs.get(k) for k in
                islice(sig.parameters.keys(), len(args),
                    ↵ len(sig.parameters)))
        )
        pickled_args = [pickle.dumps(arg) for arg in
            ↵ ordered_incoming_args]
        where_statement = ' and '.join(f'{k} = ?' for k
            ↵ in sig.parameters.keys())
        sql_select = f"""
            select _result from {f.__name__}
            ↵ where {where_statement}
        """
        c.execute(sql_select, pickled_args)
        result = c.fetchone()
        if result:
            print("Found result in Cache")
            return pickle.loads(result[0])
```

```

# assuming no result is available, add it to the
# database

result = f(*args, **kwargs)

num_inserts = ','.join('?' * (len(sig.parameters)
                                + 1))
sql_insert = f"""insert into {f.__name__}
    values({num_inserts})"""
c.execute(sql_insert, list(chain(pickled_args,
                                [pickle.dumps(result)])))
conn.commit()
return result
return wrapper
return decorator

```

```

@persistent_cache_wrapper('solutions/closest_city.sqlite')
def get_closest_city(latitude, longitude):
    time.sleep(5)
    url = 'https://llinvfuljc.execute-api.ap-southeast-'
    ↵ 2.amazonaws.com/dev/location/nearest'
    params = {'latitude': latitude, 'longitude': longitude}
    response = requests.get(url, params)
    response.raise_for_status()
    return response.json()['name']

```

79

```
get_closest_city(52.4, 4.9)
```

80

```
'Amsterdam'
```

```
get_closest_city(52.4, 4.9)
```

81

```
Found result in Cache
```

```
'Amsterdam'
```

```
get_closest_city(latitude=52.4, longitude=4.9)
```

83

```
Found result in Cache
```

```
'Amsterdam'
```

```
get_closest_city(-37.9, 145.0)
```

85

```
'Melbourne'
```

Chapter 11

Packages and modules

Packages and modules define additional functionality that extend the power of Python. This chapter explains what packages and modules are, how to find and install additional packages, how to import packages and modules, and how to create your own modules and packages.

11.1 Packages

There are packages available for everything from controlling network devices to machine learning to creating websites. As of 2020, there are over 200,000 packages available on the Python package index. The vast majority (over 99%) are freely available under a similar licence to Python itself and can be installed with a single command.

11.2 How to find packages

Given that so many packages are available for Python, how do you know which are the best?

“Awesome Python” page

The first place we recommend looking is the list of “awesome Python packages”:

<https://awesome-python.com>

This is a list contributed by many authors across many different application areas.

Python Package Index (PyPI)

A second place to look is the Python Package Index (PyPI): <https://pypi.org/>. This collection is vast and not human-curated, so the quality is variable. One guide to a package’s quality is the version number; another is the number of commits or contributors on GitHub (the most common host for the source code repositories). You can install packages from PyPI using `pip` (see below).

Anaconda.org packages

Once you have found a package and want to install it, we would recommend that you look for a pre-built package in the community Anaconda repositories, available on <https://anaconda.org>. Anaconda tends to package up only quite mature and high-quality packages in their default `anaconda` channel. In addition, there are some more current (or specialised or bleeding-edge) community channels available. The best of these is `conda-forge`.

11.3 Installing further packages

Option 1: `conda`

Installing a package is easiest when it is one of the ~600 packages supplied by Anaconda. See here for a full list:

<https://docs.anaconda.com/anaconda/packages/pkg-docs>

To install one of these into the default environment, run this from the Anaconda Prompt (Windows) or Terminal (macOS / Linux):

```
conda install package1 package2 ...
```

1

Example:

```
conda install distributed
```

1

The primary benefit of using Anaconda's small set of supported packages is that **binaries** are available for the 3 major platforms, making installation easy even for complex packages. They are also usually well tested.

Option 2: pip

The pip installer is useful for fetching the roughly 200,000 packages in the Python Package Index (PyPI) that are not available in the Anaconda repositories. This mostly works well, except for those packages written in lower-level languages (like C or Fortran) that are only distributed on source code.

Note that conda is designed to complement and work well together with pip.

Here is an example of installing a package via pip. Run this without the %%bash line from the command prompt / terminal:

```
%%bash
```

1

```
# This fetches and installs a MySQL database interface
# that is hosted on mysql.com:
pip install --allow-all-external mysql-connector-python
```

This mysql-connector-python package is an interface to the MySQL database supplied by Oracle / MySQL.)

11.4 Modules

Every Python package contains one or more modules. A module is normally a .py file: a text file with a .py extension.

To access the contents of a module you use the `import` statement:

```
import os
```

1

This gives us the ability to use functions built and stored within a module. For instance, the `os` module contains a function called `listdir` for listing a directory of files. After importing `os`, you can access its `listdir` function like this:

```
os.listdir("Data/")[:5] # the [:5] just shows the first  
↪ five, we will come back to this syntax
```

2

```
[ 'network_example.png',  
  'coil_2000.csv',  
  'xlwings',  
  'cities_only.csv',  
  'abalone.csv' ]
```

You can get a listing of all available functions and attributes in a module using `dir`:

```
dir(os)[:5]
```

1

You can get the help on something by passing it to the `help` function:

```
help(os.listdir)
```

1

In Jupyter (or IPython), you can type `?` as a shortcut:

```
os.listdir?
```

1

Additionally, we can import just a single function from a module, such as this function from Pandas for generating date ranges:

```
from pandas import date_range  
  
date_range("2017-01-01", "2017-01-31")
```

3

```
DatetimeIndex(['2017-01-01', '2017-01-02', '2017-01-03',  
                 '2017-01-04',  
                 '2017-01-05', '2017-01-06', '2017-01-07',  
                 '2017-01-08',  
                 '2017-01-09', '2017-01-10', '2017-01-11',  
                 '2017-01-12',  
                 '2017-01-13', '2017-01-14', '2017-01-15',  
                 '2017-01-16',  
                 '2017-01-17', '2017-01-18', '2017-01-19',  
                 '2017-01-20',  
                 '2017-01-21', '2017-01-22', '2017-01-23',  
                 '2017-01-24',  
                 '2017-01-25', '2017-01-26', '2017-01-27',  
                 '2017-01-28',  
                 '2017-01-29', '2017-01-30', '2017-01-31'],  
                dtype='datetime64[ns]', freq='D')
```

A standard python function called `len` can be called on any collection of objects to find out its length:

```
len(date_range("2017-01-01", "2017-01-31"))
```

1

We can also rename packages on import using `as`. This is very commonly performed, especially for modules with long names or modules nested inside packages several levels deep. (Examples of popular libraries whose module names are commonly abbreviated on import are NumPy, Pandas and Matplotlib.)

```
import numpy as np
```

5

```
np.sin(np.pi / 2)
```

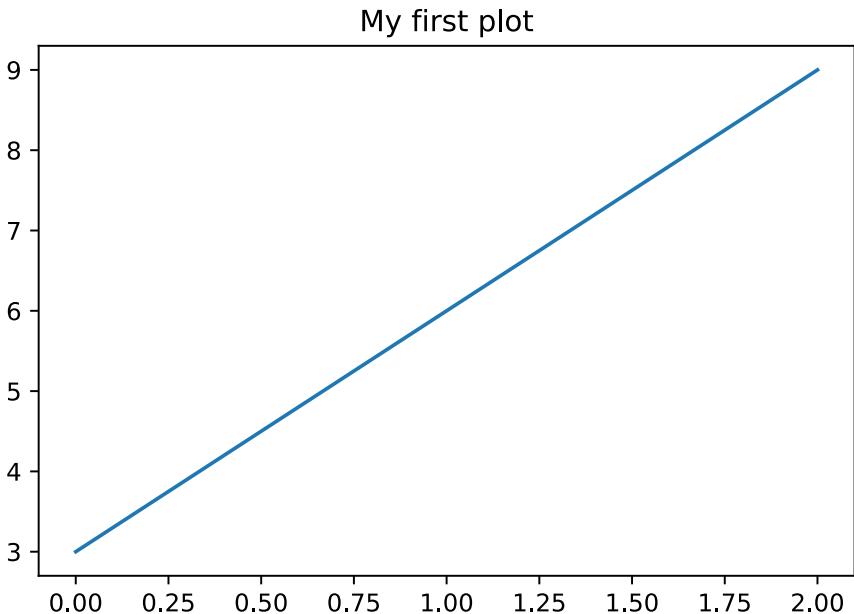
1.0

```
import pandas as pd  
  
pd.Series([1, 2, 3], index=["x", "y", "z"])
```

```
x    1  
y    2  
z    3  
dtype: int64
```

```
# This is not Python code but a special Jupyter "magic  
# command" that configures  
# Jupyter to create plots within the browser by default:  
%matplotlib inline  
  
from matplotlib import pyplot as plt  
plt.title("My first plot")  
plt.plot([3, 6, 9])
```

[<matplotlib.lines.Line2D at 0x7f2136eb59b0>]



```
import string

string.ascii_letters
```

9

```
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Exercises

1. Print out the `path` attribute in the `sys` module. This is where Python looks for modules and packages.
2. You can get a bunch of string attributes from the `string` module. For instance `ascii_letters` gives a list of the standard English characters. How many characters are there?
3. Using the `random` module's `choice` function, choose an English character at random from the list.
4. Check the help for this function using `help(random.sample)`. Can you use this function to generate a random 6 digit code? Hint: look at the other attributes in the `string` module.

```
# See solutions/modules.py
```

11

Extended Exercises

Note: Extended Exercises are for those who have Python or programming experience before, and may require external research or knowledge.

1. The `sys.modules` attribute stores the modules by name in a dictionary. Use this dictionary to directly retrieve a module object and retrieve an attribute from there.
2. You can create your own modules by putting some Python code in a file and putting that file somewhere on the path (the easiest spot is usually the current working directory). Create your own module in this way.
3. Prove that any code in a module is *executed* when it is run by putting some print statements in it. What security implications does this have?
4. To reload a module you need to use the `imp` package. Reload your module and confirm it still works.

11.5 Creating modules

Recall that a `.py` file intended to be imported into Python is called a “module”.

If a function called `convert_to_cm` were defined in a file called `unit_conversions.py`, we could use it from another script, as follows:

```
import unit_conversions
height_cm = unit_conversions.convert_to_cm(feet=6, inches=2)

type(unit_conversions)
```

1

Note that every line in the file is run upon import, so modules normally just define reusable functions, classes etc., rather than doing any actual work. This means any code outside of a function or class is actually executed!

11.6 Packages in more depth

A package is a folder containing one or more modules (.py files). Here is an example package structure:

```
sound/                      Top-level package
    __init__.py              Initialize the sound package
    formats/                  Subpackage for format conversions
        __init__.py
        wavread.py
        wavwrite.py
        ...
    effects/                  Subpackage for sound effects
        __init__.py
        echo.py
        ...
        ...
```

The special `__init__.py` files are the modules that Python runs when you import a package (e.g. `import sound`). These define the package's contents after import. These are optional on Python 3.x. (On Python 2.x they are required, but can be empty.)

11.7 Importing from packages

To import from this hypothetical sound package we can use one of these import forms:

Option 1: importing the module explicitly

```
import sound.formats.wavread
# and then:
sound.formats.wavread.wav_to_aiff('ding.wav',
                                   'ding.aiff')
```

1

Option 2: importing a single module

```
from sound.formats import wavread  
wavread.wav_to_aiff('ding.wav', 'ding.aiff')
```

1

Option 3: pulling in one or more functions

```
from sound.formats.wavread import wav_to_aiff  
  
wav_to_aiff('ding.wav', 'ding.aiff')
```

1

Chapter 12

Developing in Python

12.1 Conda environments

Conda is an environment and package manager. It was developed for Python, but part of its strength is that it can manage dependencies in any language. This includes C libraries and compilers, which many Python packages depend on. A conda environment is a little bundle of interconnected binaries and source code. They are isolated from each other, which lets you run code with different versions of dependencies, without interfering with other installed software.

You can create new conda environments with the command: `conda create -n <name> package=version [...]`, and then activate it with `source activate <name>` (Mac/Linux) or `activate <name>` (Windows). Similarly, use `source deactivate` or `deactivate` to stop using that environment.

The default environment when you install Anaconda is called the *conda root* environment.

Exercise: environments

Create a new conda environment called `minimal` with just Python 3.6. Activate it and verify that other packages you've used till now are no longer available. (For example, Jupyter.)

12.2 Getting packages

The standard Python package manager is called pip. It installs Python packages from the Python Package Index, or PyPI. You can install packages with `pip install <pkg_name>`, and update them with `pip install -U <pkg_name>`.

Conda, in addition to managing environments, can also manage the packages within those environments. It has at least two advantages over the more official pip:

- It can manage non-Python dependencies.
- It can distribute pre-compiled binaries for all the major platforms, simplifying the life of end-users.

So, although `pip install numpy` and `conda install numpy` both work, you can only `conda install boost` to get the Boost C++ library.

Use `conda install` to add new packages to the currently active environment and `conda update` to upgrade existing packages.

One downside of conda is that its repositories are not as well-populated as PyPI, but this is improving, particularly in a new community conda *channel* (like a repository) called `conda-forge`.

Many Python developers, ourselves included, have settled on using conda until it fails to find a package, then trying `conda-forge`, before falling back on pip. (Pip installs packages correctly within conda environments.)

Installing a package

One these *should* work for you (try in this order) on the command line:

```
conda install <pkg>
```

```
conda install -c conda-forge <pkg>
pip install <pkg>
```

upgrading a package

```
conda update <pkg>
pip install --U <pkg>
```

uninstalling

```
conda remove <pkg>
pip uninstall <pkg>
```

Use `pip --help` or `conda --help` for more information.

12.3 Creating and installing packages with setup.py

The standard way to make installable Python packages is to provide a `setup.py` — which is kind of like a build Makefile for Python packages. The rules for making a `setup.py` are certainly too long-winded for most users to remember. We recommend one of two approaches:

- Refer to the Python Packaging Authority's distributing guide
- Find a project you like, copy their `setup.py`.

Once you have a non-trivial Python package, it is very advantageous to write a `setup.py` file for it, because it lets you install it, even *in place* into conda environments, so that you can use it from any working directory.

Once you've created a `setup.py` file, you can:

```
pip install .
```

to install the package to your currently active environment. (Note: you might see many places recommend `python setup.py install`, but this form is deprecated.)

You can also:

```
pip install -e .
```

to install the package *from its current location* into the current environment. This means that changes to the code will propagate to the installed version, because the installed version is just a link to the current directory (with some metadata).

Finally, should you want to share your packages on PyPI, you can create a *source distribution* with:

```
python setup.py sdist
```

And then upload it with:

```
twine upload sdist/*
```

(You might need to install twine with `pip install twine`.)

12.4 Writing command-line interfaces

Python makes it very easy to write simple command-line utilities. Let's write one that finds files matching certain criteria, usable like this:

```
filefinder --min-size 100 --max-size 10000 -c .txt .
```

We can use `argparse`, which is included in the Python standard library. In addition to providing convenient parsing code, `argparse` automatically generates help and usage messages and issues errors when invalid arguments are provided.

```
import argparse
```

5

Setting up a parser

- First step for `argparse`: create parser object & tell it what arguments to expect.
- It can then be used to process the command line arguments on runtime
- Parser class: `ArgumentParser`. Takes several arguments to set up the description used in the help text for the program & other global behaviors

See <https://docs.python.org/3/library/argparse.html>

```
parser = argparse.ArgumentParser(description='Find files  
↳ matching certain criteria')
```

6

We can now add arguments to the parser:

```
parser.add_argument('-m', '--min-size', type=int,  
                   help='Minimum size, in bytes, of returned  
↳ files.')  
parser.add_argument('-M', '--max-size', type=int,  
                   help='Maximum size, in bytes, of returned  
↳ files.)
```

7

```
_StoreAction(option_strings=['-M', '--max-size'],  
↳ dest='max_size', nargs=None, const=None, default=None,  
↳ type=<class 'int'>, choices=None, help='Maximum size, in  
↳ bytes, of returned files.', metavar=None)
```

... And so on. We can then use the parser's `parse_args()` method to parse a command line. By default, `parse_args` uses `sys.argv[1:]`, but we can pass a custom list of arguments.

Exercise: making a Python command line application

- Finish writing the parser above, adding a `-c/--contains` option to find only filenames containing the given string, and a `path` positional argument specifying where to look for files.
- Create a package, `filefinder`, with a single file, `filefind.py`, containing the function below (`simple_files`), and another function, `find_files`, that uses `os.path.getsize` and `toolz.curried.filter` to find matching files in the `simple_files` stream.
- Add a function, `main`, taking no arguments, that creates the parser (as above), parses the command line, and executes the function you made.
- Add a `setup.py` file, and use the `entry_points` keyword argument to `setup()` to create command that corresponds to the `main` function.

- Use `pip install` to install your command line utility to your conda environment.

```
def simple_files(path):
    """Yield all simple files present in `path` and
    ↴ subdirectories."""
    for entry in os.scandir(path):
        if not entry.is_symlink():
            if entry.is_dir() and not entry.is_symlink():
                yield from simple_files(entry.path)
            else:
                yield entry.path
```

8

Hint: Here is the syntax for creating entry points:

```
setup(
    ...
    entry_points = {
        'console_scripts': ['command=package.module:main'],
    }
    ...
)
```

Chapter 13

Logging

Logging is important in practice. Python’s logging module is a large step up from `print()` calls: it allows you to send different levels or channels of logging messages to different destinations (via “handlers”).

The official Python docs contain two guides on logging:

- Python “Logging HOWTO”: <https://docs.python.org/3/howto/logging.html>
- Python Logging Cookbook: <https://docs.python.org/3/howto/logging-cookbook.html>

From the Logging HOWTO:

“Logging is a means of tracking events that happen when some software runs. The software’s developer adds logging calls to their code to indicate that certain events have occurred. An event is described by a descriptive message which can optionally contain variable data (i.e. data that is potentially different for each occurrence of the event). Events also have an importance which the developer ascribes to the event; the importance can also be called the level or severity.”

13.1 Examples

```
import logging  
logging.info('New user')
```

4

Notice that there is no output for `info`-level messages by default.

```
logging.warning('Disk is nearly full')
```

2

```
WARNING:root:Disk is nearly full
```

```
logging.error('Disk is full')
```

3

```
ERROR:root:Disk is full
```

To get started quickly, see the help for `logging.basicConfig`.

Here is a richer example:

```
import os  
import logging  
  
try:  
    os.makedirs('solutions')  
except FileExistsError:  
    logging.warn('solutions folder already exists')  
  
print('Writing file into solutions/ ...')
```

44

```
Writing file into solutions/ ...
```

```
/home/robert/Programs/anaconda3/lib/python3.5/site-packages/]  
↳ ipykernel/_main__.py:7: DeprecationWarning: The 'warn'  
↳ function is deprecated, use 'warning' instead
```

By creating a custom logging instance, you can also alter the format for how logging messages are displayed.

```
logger = logging.getLogger()  
  
handler = logging.StreamHandler()  
formatter = logging.Formatter('%(asctime)s - %(name)s -  
    %(levelname)s - %(message)s')  
handler.setFormatter(formatter)  
  
logger.handlers = [] # Remove old handlers  
  
logger.addHandler(handler)  
logger.warning('is when this event was logged.')
```

45

```
2016-06-15 11:40:50,734 - root - WARNING - is when this event  
    was logged.
```

```
logger.error('Disk is full')
```

46

```
2016-06-15 11:40:51,471 - root - ERROR - Disk is full
```

You can also send logging messages to a file, and loggers can have multiple handlers attached at the same time. You can set the level of logging independently as well. In this example, all messages are printed to the screen, but only errors are sent to a logging file.

```
logger.warn('This message will be displayed on the notebook  
    only')
```

1

```
logger.error("This message will be in the notebook, and also  
    in the log file")
```

1

```
!cat errors.log
```

1

Exercise

1. Review the documentation of the FileHandler to add a handler that sends log messages to a file, as well as to the Notebook. See here: <https://docs.python.org/3/library/logging.handlers.html#logging.FileHandler>
2. How would you make it so that messages *only* go to this file (and not to the notebook)?

Hint: see the help for `logging.FileHandler`

```
# See solutions/logging_example.py
```

1

13.2 Loggers, handlers, formatters, levels

Logger is where you send messages. Its name filters which messages it captures.

Loggers can have one or more handlers (what to do with the message). You might create your own handler to, e.g., send a log to a database.

Handlers have a formatter which is basically a string format for the message.

Both loggers and handlers can have “levels” which is the cutoff for which messages to capture.

Exercise: Using a JSON formatter on your log handler

Often logs are messy. Logs in the wild use many different data formats and conventions. Rather than a messy log, a great format for logs is JSON.

1. `pip install python-json-logger`
2. Import and create the log formatter:

```
from pythonjsonlogger import jsonlogger  
formatter = jsonlogger.JsonFormatter()
```

1

3. Add the formatter to the handler (`handler.setFormatter`)
4. Set the level for the handler to `logging.DEBUG`:

```
handler.setLevel(logging.DEBUG)
```

1

5. Send some log messages. Example:

```
logger.debug("A debug message")
logger.warning("A warning message")
try:
    1 + '1'
except TypeError as e:
    logger.exception(e) # to capture the traceback
```

1

13.3 Other logging packages

- `coloredlogs`: <https://pypi.org/project/coloredlogs>
- `loguru`: <https://github.com/Delgan/loguru#readme>
- `logzero`: <https://logzero.readthedocs.io/en/latest/>
- `logbook`: <https://logbook.readthedocs.io/en/stable/>

Chapter 14

Unit testing

As projects become larger the focus on reliability becomes more important. Broadly, there are two categories of errors you will encounter in your code:

1. Errors where the input to the application (whether data from a CSV file, interactions from a user of a GUI, or data requested from a remote source) is bad. These are errors you cannot prevent but they can be handled in your code with Python exceptions. In this way the application's usability is not interrupted.
2. Errors where your code is not working the way it should. These are errors you can absolutely prevent by software testing - testing your code runs under multiple scenarios and produces the output you expect.

There are four phases that must be considered in software testing:

	Phase	Tools in Python
1	Unit testing - testing discrete sections of code	<code>unittest</code> , <code>pytest</code> , <code>nose</code>

	Phase	Tools in Python
2	Integration testing - testing that the different sections of code work together correctly	tox
3	System testing - tests of the application as a whole	robotframework
4	Acceptance testing - ensuring the application meets the requirements of the business	robotframework

In this chapter we will look at unit testing in Python - the place software testing should start.

14.1 Setup

In this chapter you will use the `newster` project, found in the `extras` folder. Start by copying the `newster` directory into your home folder:

```
cp -r extras/newster ~/newster
```

1

The project has the following structure:

```
└── README.md  
└── newster.py  
└── test.html  
└── tests/  
└── urls.txt
```

You will then use the `cookiecutter` to generate a simple pip installable library with

```
cookiecutter -f gh:PythonCharmers/cookiecutter-pipproject
```

1

The project name and the app name should be `newster`. You can choose your own preference for the other options.

Once the project is created:

1. Move `newster.py` into the `newster` package folder (it will be alongside an automatically created `__init__.py` file)
2. Update the `requirements.txt` file to include the `newspaper3k` and `jinja2` libraries, and make sure to pip install them

When complete you should have a folder structure that looks like (*hint* use the `tree` command from the command line to visualise this structure):

```
└── LICENSE.md
└── MANIFEST.in
└── README.md
└── dev-requirements.txt
└── docs
    ├── Makefile
    ├── make.bat
    └── source
        ├── conf.py
        └── index.rst
└── newster
    ├── __init__.py
    └── newster.py
└── requirements.txt
└── setup.cfg
└── setup.py
└── tests
    ├── __init__.py
    └── test_sample.py
└── update_docs.sh
```

14.2 pytest

pytest (formerly known as py.test) is a simple unit-testing framework and automated test runner. It makes it easy to write test cases in Python using simple assert statements. (It also supports finding and running tests that are written in the more verbose style of the unittest framework in the Python standard library.)

You will notice that you already have a folder called `tests` in your project which contains a file, `test_sample.py`. You can run these tests from the command line with:

```
pytest
```

1

You should see that the test executes successfully. pytest looks for tests automatically when you run the `pytest` command. It traverses the directory tree looking for Python modules named either `test_*.py` (the most common) or `*_test.py`. In those modules pytest collects tests from:

- test prefixed test functions or methods outside of class
- test prefixed test functions or methods inside Test prefixed test classes (without an `__init__` method)

If you examine `test_sample.py` you will see a single function:

```
def test_pass():
    """Sample Test passing with nose and pytest"""
    assert True, "dummy sample test"
```

1

With the name `test_pass` it is collected as a test case and run automatically when you call `pytest` from the command-line.

Exercise: Writing a new test function

Ideally rather than having dummy tests that don't do anything you should write tests that test the code you've written.

1. Create a new file in the `tests` package called `test_newster.py`

2. In the file you will need to import the original newster script with
`from newster import newster`
3. Write a test function called `test_process_url` that passes a URL (choose one from the `urls.txt` file) to the `newster.process_url` function.
4. Test that the function returns a dictionary containing the keys: `url`, `title`, `date`, `img_url` and `authors` keys
 - *Hint* in pytest the most common way to write a test that passes or fails is to use an `assert` statement, e.g. `assert isinstance(data, dict)`

Run the tests using `pytest` from the command line, fixing any errors you encounter.

14.3 Parametrizing tests

Your test should now work with a single input. Often its much more useful to test against multiple inputs to ensure the output is consistent. The simple way to do this would be to use a list of URLs as input and loop over each one to test that it returned what you expected. With a bit of thinking you'll see why this is an issue:

- No matter the number of URLs you will only see one unit test in your results
- More importantly, since `pytest` uses uncaught exceptions to count whether a test has passed or failed, the first URL that fails will cause the test to stop, potentially missing errors from later URLs (and creating more work in the long run)

Instead you can “parametrize” the tests so that the same test is run by `pytest` as individual independent tests and where a failure with one URL will not cause the test to fail for all URLs.

To do this `pytest` has the `@pytest.mark.parametrize` decorator. This decorator requires two arguments (though there are other options):

1. The function arguments to be parametrized. This can be a string of comma separated argument names, or a list of string argument

- names. e.g. `'url', response'` or (for the same effect) `['url', 'response']`
2. A list or tuple of parameters, themselves stored as lists or tuples ((or `pytest.param` for more complex cases).
- Note the decorator cannot take a generator as input.
 - Note even for the case of a single argument a collection of parameters is required as `pytest` uses argument expansion to pass these parameters into the test.

Exercise: Parametrizing the URLs

Here you will update the `test_process_url` function to take parametrized URLs rather than always testing the same URL.

1. Update the `test_process_url` function to include an `url` argument.
2. Add the `@pytest.mark.parametrize` decorator to the function.
 - The input argument should be `'url'`
3. Add two URLs as the parameters:
 - `https://www.theguardian.com/environment/2020/sep/11/undraining-the-swamp-how-rewilders-have-reclaimed-golf-courses-and-waterways`
 - `https://www.theguardian.com/books/shortcuts/2015/mar/17/terry-pratchett-s-name-lives-on-in-the-clacks-with-hidden-web-code`

Rerun the test cases with `pytest`. How many tests have now been run?

14.4 Coverage

Now we have tests working, which means that we can fiddle with the code without fear of breaking things. But who tests the tests? That is, how do we know that we are writing good tests? One way to measure testing is by *coverage*, which is the lines of your code that are executed at least once by your test functions.

In `pytest`, you measure coverage by using the `--cov .` option. You can also create a per-file report of *which* lines are not covered with the option `--cov-report term-missing`.

Exercise: Measuring the amount of code tested

Measure coverage the code coverage of the above tests. *Hint* you can limit the code where coverage is measured by setting the name of the module:

```
pytest --cov=newster
```

Once the test have been run you will see a percentage of how much of your code you are actually testing. What is it?

To generate a nice report with the coverage data by running `coverage html`. This will create a series of html pages in the `htmlcov` folder. Run this report and open `htmlcov/index.html` to see a visual representation of the code that has been tested.

14.5 Property-based testing

The above tests, while providing some sanity checks, are pretty weak, because they test only one point in the possible data space. A new paradigm in testing is called *property-based* testing, which focuses on measuring *invariants* on a wide array of randomly-generated inputs.

For example, if we were writing *square* and *square-root* functions, we could test that the square of a number greater than 1 is always bigger than the number itself, and that the square root of the square of a positive number is equal to the original number (within some tolerance).

Hypothesis

Hypothesis is a Python library for property-based testing. We can use it to encode the above conditions using *strategies* for generating test cases:

```
from hypothesis import given, strategies
```

2

```
from math import sqrt

def square(x):
    return x * x
```

11

```
@given(strategies.floats(min_value=1))
def test_square(x):
    assert square(x) >= x
```

```
test_square()
```

12

So what is going on in this testing? One way to tell is to modify the test to print the value of `x`:

```
@given(strategies.floats(min_value=1))
def test_square(x):
    print(x, ',', square(x))
    assert square(x) >= x
```

13

```
# Run this:
# test_square()
```

14

As you can see, Hypothesis tests not only various random input values, but also some strategically-chosen floating point values within the constraints: the maximum floating point value, values extremely close to 1.0, values whose squares overflow to `inf`, and so on. Thus it can test a much broader range of input circumstances than ordinary test writing.

Exercise: Testing using hypothesis

The `newster` library doesn't lend itself to random input - a randomly generated URL is almost certainly going to be incorrect! Write a test that checks that the square root of the square of a positive number is itself. Does it pass Hypothesis's battery of tests?

Modify the strategy to ensure that the tests pass.

14.6 Test fixtures

It's often useful to predefine data and parameters that might be used for multiple functions. These predefined data are called fixtures. In `pytest` you can define fixtures so that the outputs are computed at different stages of the process with the scope:

- `function`: the default scope, the fixture is destroyed at the end of the test.
- `class`: the fixture is destroyed during teardown of the last test in the class.
- `module`: the fixture is destroyed during teardown of the last test in the module.
- `package`: the fixture is destroyed during teardown of the last test in the package.
- `session`: the fixture is destroyed at the end of the test session.

Some examples of fixtures might include:

- A randomly generated row from a table that is used as the input to a function. This can be at any level of scope as it shouldn't matter how often it is regenerated.
- A connection to a remote database that contains test data. This would be a session level fixture as you would not want to disconnect and reconnect in the middle of each test.
- A temporary folder where you will write output. This may be a module fixture to allow a temporary directory for all tests within the current module

A fixture is itself a function decorated with `@pytest.fixture` (for the default) or `@pytest.fixture(scope=scope)` to specify an individual scope. They can be stored along side the tests themselves, or they can be stored in a separate module. This is especially useful if you store them in `conftest.py` as `pytest` looks for this automatically and loads fixtures (and other configuration) so you do not need to import it separately.

Exercise: Adding a temporary directory fixture

1. Create a new file `conftest.py` in the `tests` directory
2. Add a new function `working_directory`. This function should:
 1. Create a new temporary directory on the system (use the `tempfile.mkdtemp` function)
 2. Use the `logging` library to log the location of the temporary directory at the `info` level
 3. `yield` the name of the temporary directory
 4. Cleanup the directory on completion of the test using `shutil.rmtree`
 5. Be decorated with the `@pytest.fixture(scope='function')` decorator so that for every call to the test function a new directory is created.

Note that by calling the script `conftest.py` pytest will autodiscover the fixtures, and therefore there is no requirement to import the `working_directory` function in to your `test_newster.py` script

3. While not necessary for this test, add a new argument `working_directory` to the `test_process_url` function.
 - *Hint* This must match exactly with the name of the fixture function.
4. Run the tests with `pytest --log-cli-level=INFO`

14.7 Mocking the results of a function

Of course the fixtures can go much further. For example within a fixture you can temporarily overwrite (or “monkey-patch”) the results of a function (known as “mocking”). This is extremely useful for testing functions that rely on the response of another function.

Most importantly mocking should be used to separate tests for your code from reliance on external libraries / resources. For example if your test for `newster` is going to call a URL many times via the `newspaper3k` library it is unnecessary for you to test this multiple times (this would be integration testing).

Exercise: monkey-patching a function result

Out of the box pytest includes a `monkeypatch` fixture designed to overwrite the value of a function temporarily and then revert once the function has finished.

1. Write a new `test_generate_report` function in `test_newster.py`.

It should:

1. Use the fixtures `working_directory` and `monkeypatch`
2. Define a new local function `mockreturn`:

```
def mockreturn(url):
    return {'url': url, 'title': 'Title', 'date':
        '23-09-2020', 'img_url': None, 'authors':
        ['Henry', 'Ed']}
```

3. Use the `monkeypatch` fixture to overwrite the `process_url` function:

```
monkeypatch.setattr(newster, 'process_url',
    mockreturn)
```

4. Use the `monkeypatch` fixture to change directories to the working directory:

```
monkeypatch.chdir(working_directory)
```

5. Run the `newster.generate_report` function with:

- `urls=['http']`
- `output=open('report.html', 'wt')`

6. Assert that `report.html` exists and the file size is greater than zero

- *Hint* look at `os.path.exists` and `os.stat`

2. Once again run the tests with `pytest --log-cli-level=INFO`

3. Check that the temporary directory shown in the logs doesn't exist any longer

4. Run `pytest --cov=newster`. What percentage of the module is included with the unit test?

5. Parametrize the test so that it tests both the HTML and CSV report formats, and run pytest with coverage again. Have you tested 100% of the code in this module?

Mocking with `pytest-mock`

The previous test overwrote one of the functions we had defined. Python (as of version 3.3) has a built in `unittest.mock` module which is designed to let you overwrite specific objects. The `pytest-mock` extension allows you to easily integrate this with your `pytest` tests by providing a `mocker` fixture you can include in your tests.

In your test function you can include a `mocker.patch` call to patch a Python object. For example:

```
mocker.patch('newspaper.Article', autospec=True)
```

1

This patches `newspaper.Article` by replacing it with a `unittest.mock.MagicMock` object. *Note* the container object is required, so if you are patching an object in the same file you're working from you will either need to reference `__main__.functionname` (if you're just running the mock tests), or the full module location of the object `path.to.object`. This is not advised.

The `autospec` keyword argument is extremely powerful in mocking objects to be used inside functions. From the documentation:

If you set `autospec=True` then the mock will be created with a spec from the object being replaced. All attributes of the mock will also have the spec of the corresponding attribute of the object being replaced. Methods and functions being mocked will have their arguments checked and will raise a `TypeError` if they are called with the wrong signature. For mocks replacing a class, their return value (the ‘instance’) will have the same spec as the class.

A good example of why this is useful would be to compare these two calls to the `Article` object:

```
article = Article(url)
article.download()
```

1

versus

```
article = Article(url)
article.downloa()
```

1

On first inspection these look extremely similar. However the second version is missing the d at the end of download. If you don't set a specification when you are mocking an object using `unittest.mock` then any function call or attribute accessor on that function will be valid, and the second version would pass your unit test. By having `unittest.mock` build the specification (with `autospec=True`) then only the first version of the above would pass.

There is one more trick when dealing with `autospec`: if the attributes are defined as part of the `__init__` function they will not be extracted by the `autospec`. Instead subclass the object to define required attributes at the top level and use it in place of the specification:

```
from datetime import date
class ArticleForTest(newspaper.Article):
    publish_date = date.today()
    authors = []
    url = ''
    title = ''
    top_image = ''
mocker.patch('newspaper.Article', autospec=ArticleForTest)
```

1

Exercise: mocking with `pytest-mock`

1. Create a new test in `test_newster.py` called `test_process_url_with_mock` that takes the `mocker` fixture as an argument.

2. Inside the function define the class ArticleForTest as above.
3. When mocking instead of using `mocker.patch` directly, use:

```
mocker.patch.object(newster, 'Article',  
    ↵    autospec=ArticleForTest)
```

1

This patches the article object as imported in the `newster.py` script

4. Run the `process_url` function with two URLs:

- '`bad_url`'
- '`https://www.theguardian.com/environment/2020/sep/11/understanding-the-swamp-how-rewilders-have-reclaimed-golf-courses-and-waterways'`

Does the test pass?

Chapter 15

Intro to test-driven development

Test driven development is a technique where you write the tests for your code before you write your code. This technique when applied in the right circumstances has a very well proven industry track record in terms of improving quality of code and reducing overall defects count and as such is an important technique to have in your repertoire.

15.1 Why would we want to write tests first?

One of the most jarring things about test driven development for people that are not used to it is the act of writing out the test first.

This mindset is not something that tends to come naturally at first (it didn't for me initially) because most people are used to writing the code first and get an introduction to the software engineering practice of testing later on in their journey of learning to develop software.

The main reason that firms will go to the effort and expense of hiring a dedicated team for testing or QA that is completely separate to the team making the product is because it's actually quite hard to test things that you have made. The mindset of creation is not the same one that really helps for test-

ing of trying to pull everything apart.

To effectively test a code base involves a mindset of mercilessly pulling everything apart, looking for any edge cases that lead to bugs, looking for that one in a million bug that rarely comes up.

However it goes a little beyond this, when we are testing our own code some biases come into play:

Confirmation Bias

Definition: The tendency to search for, interpret, focus on and remember information in a way that confirms one's preconceptions.

It can be hard to search for ways in which your program does not work, even if that's what you want to do this bias can creep in and make it hard for you to see the ways in which you can make your code break, even if that's what you are trying to do!

If you write the tests first: You don't have an implementation yet that will break, so it's easier to write tests that will break your future code without running into this cognitive bias. Because the code doesn't exist yet you don't have a chance to get swayed by your existing preconceptions.

Testing the implementation details

When you are writing some code you are very intimately aware of how it will generate the results you want.

When we have effective tests we are really just wanting to test the interface and properties of how the code transforms inputs to outputs. *How* exactly the code does this is not what we want to test, the implementation of the code may be changed later and a sizable part of good code architecture is enabling changes in implementation to not break existing code that uses it. Knowing more about the code implementation can in some cases actually make it harder to write a good test!

For example let's say you have a situation where you have a function where you are trying to multiply two numbers together.

Let's say the way it does it is to call a web API for a calculation service that

calculates the numbers on a server and returns them back to you. You might decide to test that the function worked by logging in to the service to see if the API call was successful and see if the return value matched what you expected.

While this may test your function adequately it's testing it via the incidental way in which it is implemented, what if this changes later? What if we want to move to a different service or different approach? If we have our tests written this way we can have our tests end up failing even when the function keeps working perfectly!

If you write the tests first: there are no implementation details yet. So it's much harder to fall into the trap of testing the implementation instead of the interface.

The tests can't be skipped

For a variety of reasons pressure might come up to skip tests entirely. Sometimes this is appropriate but very frequently it is not. The internal quality of a project is not something that can be traded off for speed (<https://martin-fowler.com/bliki/TradableQualityHypothesis.html>) because internal quality of your codebase is the enabler of speed of development of features.

By writing the tests first people will not skip out on testing, this can be a good practice for junior engineers who haven't built up the judgment to know where tests can be skipped.

Also for organizational reasons it might be much more preferable to write the tests first if there is a lack of procedural effectiveness that can get in the way of the software development process. For example if there is a large amount of churn of personnel on a particular project there's a substantial risk that things such as tests and documentation will never get written if it's not done incrementally along with the code. Coming back up to speed on these things is a substantial productivity killer because the lower the gap in time and gap in communications there is between writing the code and tests the easier it is to do.

15.2 How to write the tests first: Red, Green, Refactor

There are many ways that people go about writing tests for their code. Sometimes people write tests after they have some functionality because they know something that the code should do and they wish to ensure they do it. Another approach is to write the tests before you have written the code, there is a workflow that is highly effective for this called “red, green then refactor”.

This technique comprises the following 3 steps:

1. Red: Write a test that will both test your code appropriately and will fail when you run it
2. Green: Implement the minimal amount of code to make the test you just wrote pass
3. Refactor: Improve the code from the second step

The steps in the process here are quite important and it's important that they are 3 distinct steps. We will break this down step by step with an example where we are making a function that will multiply two numbers together.

15.3 Step 1: Red

The first step when considering a test is what you expect the output to be for some given input.

This will help you clarify what the interface of your functionality is and what the requirements are.

We will use pytest for these examples, if you don't have pytest installed yet you can install it with:

```
pip install pytest
```

1

The first step is to make a test that just tries to call the functionality. In this case we want to see if we can import the module/function.

```
# test_multiply.py
def test_multiply_function():
    from tdd_example import multiply_two
```

1

Because the function does not exist yet this will fail.

```
(master)$ python -m pytest .
=====
test session starts
=====
platform linux -- Python 3.7.2, pytest-3.10.1, py-1.6.0,
    pluggy-0.7.1
rootdir: /home/janis/TDD-example, inifile:
plugins: remotedata-0.3.0, openfiles-0.3.0,
    doctestplus-0.1.3, arraydiff-0.2
collected 1 item

test_multiply.py F
    [100%]

=====
FAILURES
=====
----- test_multiply_function -----
-----
def test_multiply_function():
>     from tdd_example import multiply_two
E     ModuleNotFoundError: No module named 'tdd_example'

test_multiply.py:2: ModuleNotFoundError
=====
1 failed in 0.03 seconds
=====
```

This is good, as it shows that our test is failing.

Let's make a stub function now but deliberately not try to do anything with it:

```
# tdd_example.py  
def multiply_two(first, second):  
    """Multiply the two arguments together"""  
    raise NotImplementedError
```

1

Now when we run the test again we get the following:

```
(master)$ python -m pytest .  
===== test session starts  
↳ ======  
platform linux -- Python 3.7.2, pytest-3.10.1, py-1.6.0,  
↳ pluggy-0.7.1  
rootdir: /home/janis/TDD-example, configparser:  
plugins: remotedata-0.3.0, openfiles-0.3.0,  
↳ doctestplus-0.1.3, arraydiff-0.2  
collected 1 item  
  
test_multiply.py .  
↳ [100%]  
  
===== 1 passed in 0.02 seconds  
↳ ======
```

1

Now this does what the test case needs and passes, but there's the issue that the function doesn't actually do anything useful yet! Despite this we are making progress, we know we can import the function, we just have to implement it.

At this point it's important we think about the function requirements and make sure we figure out some tests that will indicate the function works AND that make the tests go red again.

```
# test_multiply.py  
def test_multiply_function():  
    from tdd_example import multiply_two
```

1

```
assert multiply_two(1, 4) == 4
assert multiply_two(0, 3) == 0
```

Now when we run this:

```
=====
test session starts
=====
platform linux -- Python 3.7.2, pytest-3.10.1, py-1.6.0,
    pluggy-0.7.1
rootdir: /home/janis/TDD-example, inifile:
plugins: remotedata-0.3.0, openfiles-0.3.0,
    doctestplus-0.1.3, arraydiff-0.2
collected 1 item

test_multiply.py F
    [100%]

=====
FAILURES
=====
----- test_multiply_function
-----
def test_multiply_function():
    from tdd_example import multiply_two

>         assert multiply_two(1, 4) == 4

test_multiply.py:4:
-----
    first = 1, second = 4

        def multiply_two(first, second):
            """Multiply the two arguments together"""
>         raise NotImplementedError
E         NotImplementedError
```

```
tdd_example.py:3: NotImplementedError  
===== 1 failed in 0.07 seconds  
↳ =====
```

Now we get a failure from the test we just added to, this is what we want since we haven't implemented the multiplication functionality just yet.

Why it's important that the test fails at the red step

You'll have noticed before that there's some situations where the test passes despite the functionality not being entirely there, you want to make sure in any cases like this you change your tests to make sure that missing functionality fails. If your tests pass without the code being implemented it means that your test cases don't actually have any power, you don't want to have to wonder if the test passes because of some incidental factor.

So if any test passes at the red step before there's implementation, this is an issue. You'll want to build a habit of fixing up any bad test cases as soon as possible. Bad test suites in the worst case this can actually give you false confidence that your program is working when in fact it is not.

Having the test fail initially is important because the feedback you get when you implement in the next step (green) will then be much more meaningful.

15.4 Step 2: Green

This is the step where we write the code to make our functionality work. In some workflows people are used to doing this step first. The fact that this step is *not* first is the defining characteristic of Test Driven Development, the test is the initial driver of the development in this mindset, not the other way around.

In this step it is important to not be perfectionistic, you want to do the absolute minimum amount of coding to make your test cases pass. There's 2 reasons for this:

1. Firstly you want to make sure that you can meet your requirements as stated as fast as possible
2. When you improve your code it is helpful to have a test case that will show you if your improvements have actually broken the test case.

In this case the implementation to get the tests green is fairly straightforward:

```
# test_multiply.py
def multiply_two(first, second):
    """Multiply the two arguments together"""
    return first * second
```

Let's run the test case:

```
(master)$ python -m pytest
=====
test session starts
=====
platform linux -- Python 3.7.2, pytest-3.10.1, py-1.6.0,
    ↵ pluggy-0.7.1
rootdir: /home/janis/TDD-example, configparser:
plugins: remotedata-0.3.0, openfiles-0.3.0,
    ↵ doctestplus-0.1.3, arraydiff-0.2
collected 1 item

test_multiply.py .
    ↵ [100%]

=====
1 passed in 0.02 seconds
=====
```

Now we have a passing test!

Less code is better

In general, by doing the minimal amount necessary to make the test pass, you give yourself a prototyping opportunity. This step can give you insight into the best way of doing things. By making a prototype of sorts you get to see

not only how to make an improved version but whether you need to improve it at all.

Sometimes what you get in step 2 ends up being good enough and you can just move on to more productive tasks. But when step 2 is not good enough it's much easier to use the knowledge you gained in the actual act of doing it to improve your code than it is to hypothesize about it beforehand.

It turns out that hypothesizing beforehand isn't always even possible because you don't know what you will learn in the process ahead of time.

(There is a great book by Kenneth O'Stanley called “Why greatness cannot be planned: the myth of the objective function”. This explains the mechanism by which this iterative approach to learning allows you to produce things that are not possible if you attempted to work at an objective from the start. This is because you get to learn along the way, including things you didn't anticipate ahead of time.)

15.5 Step 3: Refactor

This is the step where you can improve the code in various ways.

This is a great time to reduce the amount of technical debt that your code has, you have a test case or a few test cases that will inform you if you make a mistake and this will give you instant feedback on latest changes that can uncover bugs extremely efficiently.

Let's say someone tries using the function with a string:

```
>>> from tdd_example import multiply_two
>>> multiply_two("abc", 2)
'abcabc'
>>> multiply_two("4", 2)
'44'
```

1

Perhaps we don't want our function to allow strings as input for some reason. In this situation we may decide to change the function to raise a `TypeError` if a string is provided.

In this case we would add a new test case that catches this issue:

```
# test_multiply.py
def test_multiply_function():
    from tdd_example import multiply_two

    assert multiply_two(1, 4) == 4
    assert multiply_two(0, 3) == 0

def test_multiply_function_no_strings():
    """Test that attempting to use the multiply function with
    a string as an argument raises TypeError"""
    from tdd_example import multiply_two

    import pytest
    with pytest.raises(TypeError):
        multiply_two("abc", 2)
```

Now when we run this:

```
(master)$ python -m pytest
=====
test session starts
=====
platform linux -- Python 3.7.2, pytest-3.10.1, py-1.6.0,
    pluggy-0.7.1
rootdir: /home/janis/TDD-example, inifile:
plugins: remotedata-0.3.0, openfiles-0.3.0,
    doctestplus-0.1.3, arraydiff-0.2
collected 2 items

test_multiply.py .F
    [100%]

=====
FAILURES
=====
----- test_multiply_function_no_strings
-----
```

```
def test_multiply_function_no_strings():
    """Test that attempting to use the multiply function
       with
       a string as an argument raises TypeError"""
    from tdd_example import multiply_two

    import pytest
    with pytest.raises(TypeError):
>        multiply_two("abc", 2)
E        Failed: DID NOT RAISE <class 'TypeError'>

test_multiply.py:14: Failed
=====
1 failed, 1 passed in 0.06 seconds
=====
```

We see that we have our original test still passing, and we will want to make sure we keep it passing. However the newer test is not passing yet which is what we want.

We can implement this functionality now in the function:

```
def multiply_two(first, second):
    """Multiply the two arguments together"""
    if isinstance(first, str) or isinstance(second, str):
        raise TypeError("Strings are not allowed")
    return first * second
```

Now when we run the tests:

```
(master)$ python -m pytest
=====
test session starts
=====
platform linux -- Python 3.7.2, pytest-3.10.1, py-1.6.0,
  pluggy-0.7.1
rootdir: /home/janis/TDD-example, configparser:
plugins: remotedata-0.3.0, openfiles-0.3.0,
  doctestplus-0.1.3, arraydiff-0.2
```

```
collected 2 items

test_multiply.py ..
    ↳ [100%]

===== 2 passed in 0.02 seconds
↳ =====
```

The important part of the refactor step is to make improvements to the code that do not break the tests that were passing before. When you get used to working in this fashion where you make tests first then functionality second it can help you feel more confident that your newer changes won't break older code.

TDD exercise

We will run through writing some new functionality in a TDD manner.

15.6 Best practices: exercise with TDD and decorators

Decorators are useful for factoring out repeated boilerplate code from functions. One of many uses is to perform validation and preprocessing of input arguments to simplify and streamline the implementation of functions.

In this example (from the Starborn visualization package: <https://github.com/Python-Charmers/starborn>), we write a decorator that allows functions to accept synonyms for keyword arguments. (Useful for simply supporting multiple calling conventions or while refactoring an API.)

Goal: write a decorator `map_kwargs` that can be applied like this:

```
@map_kwargs({'hue': 'color', 'colour': 'color'})
```

to a function like this:

```
@map_kwargs({'hue': 'color', 'colour': 'color'})  
def violinplot(*args, **kwargs):  
    assert 'hue' not in kwargs and 'colour' not in kwargs
```

so that the decorated function: 1. accepts only one of the arguments (mutually exclusive); 2. receives normalized argument names according to the dictionary mapping (here, to the US English spelling of “color”).

Exercise 1: write some tests for this using **pytest**

```
# %load solutions/test_decorator_map_kwargs.py
```

12

Exercise 2: implement the **map_kwargs** decorator function

```
# %load solutions/decorator_map_kwargs.py
```

13

Chapter 16

Files and paths

There are multiple ways that you can represent file system paths. Dealing with file paths correctly involves a bit of planning, especially if you intend for your code to be cross platform.

Thankfully since Python 3.4 the `pathlib` module has made this substantially easier to do. (If you have to support Python 2 there is also backported library called `pathlib2` which will provide these features.)

16.1 Difficulties representing paths as strings

If you store paths in strings, a number of things can go wrong.

Backslashes

Several problems occur because backslashes in strings introduce escape sequences.

As one example, this is a `SyntaxError` (in Python 3.x):

```
path = 'C:\Users\d123456'
```

1

Why? Python expects `\U` to be followed by a sequence of numbers to repre-

sent an arbitrary unicode character.

In this next example, \n is a newline and \t is a tab character:

```
path = 'C:\Documents and settings\norman\todo_list.txt'  
print(path)
```

2

```
C:\Documents and settings  
orman          todo_list.txt
```

Using “raw” strings

Use an r prefix to designate a string as a “raw string” – telling Python not to interpret backslashes as having a special meaning. This fixes the problem of escaping:

```
path = r'C:\Users\d12345'  
print(path)
```

1

```
C:\Users\d12345
```

```
path = r'C:\Documents and settings\norman\todo_list.txt'  
print(path)
```

3

```
C:\Documents and settings\norman\todo_list.txt
```

Beware that adding a trailing backslash character to the end of a string still requires special handling:

```
path = r'C:\Users\norman' + '\\'  
print(path)
```

4

```
C:\Users\norman\
```

Raw strings are useful, especially for entering Windows paths and regular expressions. However, there are still multiple issues.

Path differences across platforms

Differences in the path separator across platforms can also cause issues. For example:

```
auto_path = r"Data\Auto.csv"

# Works on Windows but fails on Mac / Linux:
import pandas as pd
pd.read_csv(auto_path)
```

37

16.2 Path operations with `pathlib`

Using `pathlib` is now the recommended way of dealing with paths in Python. It solves a number of the issues above by providing a dedicated `Path` data type distinct from strings.

```
import pathlib

data_path = pathlib.Path(".") / "Data"
data_path
```

30

```
PosixPath('Data')
```

`pathlib` contains a number of convenience functions for manipulating paths:

```
auto_path = data_path / 'Auto.csv'
csv = auto_path.read_text(encoding='utf-8')
```

45

```
# Convert to absolute path
data_abs = data_path.absolute()
data_abs
```

46

```
PosixPath('/home/janis/PythonCharmers/NetworkEngineers/Data')
```

```
# Find extensions
config_file = pathlib.Path("config.ini")
config_file.suffix
```

50

```
'.ini'
```

```
config_file.stem
```

51

```
'config'
```

It may be useful to get the current user's home directory – this is usually where their files are kept. This can be done through the `.expanduser` method:

```
data_path = Path('~/Data').expanduser()
data_path
```

4

```
PosixPath('/Users/schofield/Data')
```

Wildcard expansion (globbing)

Path objects allow handy wildcard expansion (“globbing”) for files and directories. Here is an example:

```
files = data_path.glob('*.*csv')
filelist = list(files)
filelist[:5]
```

2

```
['./Data/states_only.csv',
 './Data/glass.csv',
 './Data/abalone.csv',
 './Data/ocean.csv',
 './Data/coil_2000.csv']
```

This works recursively too:

```
morefiles = data_path.glob('**/*.csv', recursive=True)
filenames[::6]
```

```
./Data/Rainfall/10011.csv
./Data/Rainfall/10010.csv
./Data/Rainfall/10012.csv
./Data/Rainfall/10013.csv
./Data/linear_bucket/test_data.csv
./Data/BayesianDataAnalysis/tulips.csv
```

Exercise

Using `glob`, print out any filename that starts with an “a”.

Hint: You’ll need to use `string.split()` to get just the filename, along with negative indexing.

Converting Path objects to strings

All standard library functions in recent Python versions (3.6 onwards) that accept file paths should operate with path-like objects.

If you do encounter an older or buggy 3rd-party library that requires a string to represent a file path, use the `os.fspath()` function to convert your path to a string. See PEP 519 for details.

16.3 Reading text files

Text files are just binary files whose contents are valid “text” in some encoding. The most common text encodings for files are UTF-8 and ASCII. Other less common ones include UTF-16, Latin-1, and CJK.

The easiest way to read a text file’s contents in Python is by calling the `read_text()` method on the `Path` object:

```
airfares_path = data_path / 'airfares.txt'
airfares = airfares_path.read_text(encoding='utf8')
```

```
print(airfares[:72])
```

ABE ATL 613.84 3627.18 692 215.77 102.17 188.12
↪ 87.06

Best practice is to specify the encoding always, to help make your code explicit and portable across platforms with different default encodings.

16.4 Reading binary files

Binary files are any file that isn't stored as text, including compiled programs, Word documents, and database files. We can read binary files in and print out the results as follows:

```
testdata_path = data_path / 'testdata.bin'  
data = testdata_path.read_bytes()  
print(data[:50])
```

57

b'SQLite format 3\x00\x04\x00\x01\x01\x00@
↳ \x00\x00\x00.\x00\x00\x00\x04\x00\x00\

The result of calling `.read_bytes()` on a `Path` is a `bytes` object. What is printed includes the data in the file and information about the structure. For example, the first few characters tell us it is a SQLite database, specifically using format 3. We can use this information to look-up the structure.

We may also want to just retrieve the strings from this file. We can do that by looking for groups of printable characters.

To explore the strings in this binary file further, we need to convert our bytes into a string, which we can do like this:

```
string_data = data.decode('ascii', errors='ignore')  
string_data[:50]
```

50

'SQLite format 3\x00\x04\x00\x01\x01\x00@
↳ \x00\x00\x00.\x00\x00\x00\x04\x00\x00

```
from string import printable

current_word = ""
words = []
for letter in string_data:
    if letter in printable:
        current_word += letter
    else:
        if len(current_word) > 3:
            words.append(current_word)
            current_word = ""

if len(current_word) > 3:
    words.append(current_word)

words
```

61

```
['SQLite format 3',
 '@ .',
 '.-\rZ',
 'sZT%',
 'sindexdocument_idsdocuments',
 'CREATE UNIQUE INDEX document_ids ON documents (did)@',
 'Otabledocumentsdocuments',
 'CREATE TABLE documents (\n\tid INTEGER NOT NULL, \n\ttext
 ↵ TEXT, \n\tuploaded DATETIME, \n\tuid INTEGER, \n\tPRIMARY
 ↵ KEY (id), \n\tFOREIGN KEY(uid) REFERENCES users
 ↵ (uid)\n',
 'stableusersusers',
 'CREATE TABLE users (\n\tid INTEGER NOT NULL, \n\tusername
 ↵ TEXT, \n\tpassword TEXT, \n\tprofile TEXT, \n\tPRIMARY
 ↵ KEY (id)\n)\r',
 '*****',
 'x$Auser3{"freqs": {}, "weight": 2}\$',
 'Auser2{"freqs": {}, "weight": 2}\$',
 'Auser1{"freqs": {}, "weight": 2}\r',
 'c.33A',
```

```
'Document 1 by user32015-05-20 01:46:26.134518',
'33ADocument 0 by user32015-05-20 01:46:26.083578',
'33ADocument 1 by user22015-05-20 01:46:26.039238',
'33ADocument 0 by user22015-05-20 01:46:25.989405',
'23A\tDocument 1 by user12015-05-20 01:46:25.9264112',
'3A\tDocument 0 by user12015-05-20 01:46:25.860113\n']
```

If we know more about the structure, we can load the binary file using a library that supports it. For instance, we now know this is a SQLite database, so we can load it as such.

String searching can also be effective when the file has been corrupted. In `Data/corrupted.bin` we have corrupted the same file with some random data.

Exercise

Read the strings from the corrupted file and see what was lost.

16.5 Compressed data (`zip`, `gzip`, `bz2`, ...)

Python's standard library has modules for in-memory compression and decompression of data in different formats: `gzip`, `zip` (PKZip), `lzma`, `bzip2`, and Unix `tar`.

Here is an example of transparently decompressing data from a `gzip`-compressed file:

```
alice_gz_path = data_path / 'alice_in_wonderland.txt.gz'

import gzip
alice_data = gzip.GzipFile(alice_gz_path).read()
alice_data[:55]
```

6

```
b'\xef\xbb\xbfThe Project Gutenberg EBook of Alice in
Wonderland, '
```

Notice that the result of calling `.read()` on the `GzipFile` is a bytes object (always) with the uncompressed data. If this represents text in some

encoding (like UTF-8), you can decode it to a `str` object as follows:

```
alice_text = alice_data.decode('utf-8')
type(alice_text)
```

8

```
'The Project Gutenberg EBook of Alice in Wonderland, by'
```

In case you are wondering, `\ufeff` is the “byte-order mark” often present in text files on Windows.

The other modules `bz2`, `lzma`, etc. have similar interfaces to `gzip` for their respective compression formats.

Exercise

Open the `Data/500-worst-passwords.txt.bz2` file and list the first 10.

```
# See solutions/badpasswords.py
```

11

Zip files

Zip files work a little differently, in that they contain *collections* of files, not just a single file. The `zipfile` module works with zips:

```
import zipfile

data_path.glob('*.*zip')[:6]
```

11

```
['Data/housesalesprediction.zip',
'Data/twitter.json.zip',
'Data/LoanStats3a.csv.zip',
'Data/spambase.zip',
'Data/world-university-rankings.zip',
'Data/USStatesData.zip']
```

```
states_path = data_path / 'USStatesData.zip'
z = zipfile.ZipFile(states_path)
z.filelist
```

13

```
[<ZipInfo filename='States.csv' compress_type=deflate
    ↵ external_attr=0x20 file_size=3797 compress_size=1791>,
<ZipInfo filename='States.json' compress_type=deflate
    ↵ external_attr=0x20 file_size=10875 compress_size=2060>,
<ZipInfo filename='States.xlsx' compress_type=deflate
    ↵ external_attr=0x20 file_size=15239 compress_size=11856>,
<ZipInfo filename='States.xml' compress_type=deflate
    ↵ external_attr=0x20 file_size=9834 compress_size=2108>]
```

```
z.extractall("output_folder") # Helper function to simply
    ↵ extract all files to a directory

file = z.getinfo("States.csv")
s = z.extract(file)
s # Gives filename
```

19

```
'/root/home/robertlayton/pythoncharmers/DataAnalytics/'
    ↵ modules/2070 Standard library
    ↵ tour/States.csv'
```

```
buffer = z.open(file)
buffer.read()[:50] # Read files without needing to unzip to
    ↵ file
```

31

b'name,abbreviation,capital,most populous city,popul'

Exercise

Write a function `read_csv_from_zip(zip_filename, name)` which reads and returns the given file name from inside a zip file like `huge_us_stocks_data.zip` as a DataFrame.

See `solutions/read_csv_from_zip.py`

16.6 Low-level file operations

If you are not using the simple `.read_text()` or `.read_bytes()` methods of `pathlib` or another library like *Pandas* for reading files, you can operate on file objects manually. This is lower-level, but is useful sometimes, such as for processing enormous log files (e.g. terabytes) line-by-line.

Before reading a text file's contents the low-level way in Python, you open the file in *text* mode. One way is as follows:

```
file = open(airfares_path, mode='rt') # rt means "read as
    ↵ text"
# Code to read the file's contents goes here
...
file.close()
```

1

In place of the ellipsis `...` you add code to read some or all of the file. You can read an entire text file into a string in Python as follows:

```
contents = file.read()
```

1

Or, if the file is large, you can read the file one line at a time by iterating over the file:

```
for line in file:
    # Code to process the line goes here:
    ...

```

1

To open a file for reading in *binary* mode, use the '`'rb'`' flags for the `mode` keyword argument:

```
file = open(airfares_path, mode='rb') # rb means "read as
    ↵ binary / bytes"
# Code to read the file's contents goes here
...
file.close()
```

1

The result of calling `.read()` on a file opened in binary mode is a bytes object.

Automatically closing files

Files in Python are “context managers”; they can be used with Python’s special `with` keyword to close themselves automatically at the end of a block of code. Best practice is therefore to read files like this:

```
with open(airfares_path, mode='rt') as file:  
    contents = file.read()
```

Notice that there is no need for `file.close()`. The file will be closed automatically at the end of the `with` block.

The low-level way to read the binary file from earlier is:

```
testdata_path = data_path / 'testdata.bin'  
with open(testdata_path, 'rb') as inf:    # rb means "read as  
    ↪ bytes"  
    data = inf.read()  
print(data[:50])
```

58

b'SQLite format 3\x00\x04\x00\x01\x01\x00@
↳ \x00\x00\x00.\x00\x00\x00\x04\x00\x00\

Low-level operations on compressed files

First, observe the result of using Python's built-in `open` function to read the first 50 bytes of a `gzip`-compressed file:

```
alice_gz_path = data_path / 'alice_in_wonderland.txt.gz'  
with open(alice_gz_path, mode='rb') as f:  
    print(f.readline()[:50]) # first 50 bytes
```

6

```
b'\x1f\x8b\x08\x08[3]0\x00\x03alice_in_wonderland.txt\x00\xcd\xfd\xdb\r\n' + '\r\n\x8a\xbe+B\xff\x90\xd4\x0b'
```

The result is incomprehensible compressed binary data. However, there is a corresponding open function in the `gzip` module that decompresses the file transparently in memory:

```
import gzip
with gzip.open(alice_gz_path, mode='rt') as f:
    print(f.readline()) # first line
```

7

ALICE'S ADVENTURES IN WONDERLAND

CPUs and memory are fast; disks are slow. It is considerably faster to decompress a large file on the fly in memory than to decompress it into a file on disk only to re-read it from disk back into memory (unless you will read it many times).

16.7 Cryptographic hashing (optional topic)

A cryptographic hash is a fingerprint of some binary data like a file's contents that is (for practical purposes) unique. It can be used to verify the integrity of a file or blob of data: if the hash of two files is the same, the probability that the files are identical is very high, so one of them is extremely unlikely to have been corrupted or tampered with.

Common cryptographic hash functions include MD5, SHA1, and SHA256.

This example shows how to compute the MD5 hash of a file's contents

```
import hashlib

portfolio_path = data_path / 'portfolio_flat.h5'
data = portfolio_path.read_bytes()

myhash = hashlib.md5(data)
myhash.hexdigest()
```

37

'4044e2839de128940993f91c6a8a778a'

```
# Linux:  
# !md5sum Data/portfolio_flat.h5  
# macOS:  
# !md5 Data/portfolio_flat.h5  
# Windows: not available
```

41

```
MD5 (Data/portfolio_flat.h5) =  
↳ 4044e2839de128940993f91c6a8a778a
```

Computing a hash is a quick way to test if two byte-strings are equivalent, as hashes of very long byte-strings will still be short (commonly 128 to 512 bits).

```
mystring = "The cat in the hat"  
  
print(hashlib.md5(mystring.encode()).hexdigest())
```

42

```
962c41d7eb82948eb42b756bf0e74bd8
```

Challenge exercise

Combine your ability to search for files with cryptographic hashing to find duplicate files in the Data directory.

Hint: get the hashing part working first, and create a helper function that takes a filename as input and returns as hash. Then work with a dictionary to store the mapping between a hash and a list of corresponding filenames.

Hint: Check out the `groupby` function in the `toolz` package.

Chapter 17

Working with IP addresses

Python provides a number of helpful utilities for dealing with IP addresses in the standard library module `ipaddress`. There's a number of 3rd party tools that can also help with these tasks.

17.1 Validating IP Addresses

```
import ipaddress
```

12

```
try:  
    addr = ipaddress.IPv4Address("3000.200.100.0")  
except ipaddress.AddressValueError:  
    print("Invalid address")  
else:  
    print("Valid address")
```

18

Invalid address

17.2 Finding your local IP address

There are a few ways you can find your local IP address.

*nix

ifconfig

Windows

ipconfig

Typical output:

```
$ ifconfig  
br-0fc40b53eac2 Link encap:Ethernet HWaddr  
02:42:9a:57:42:94  
      inet addr:172.18.0.1 Bcast:172.18.255.255  
Mask:255.255.0.0  
          UP BROADCAST MULTICAST MTU:1500 Metric:1  
RX packets:0 errors:0 dropped:0 overruns:0 frame:0  
          TX packets:0  
errors:0 dropped:0 overruns:0 carrier:0  
          collisions:0 txqueuelen:0  
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)  
  
docker0    Link encap:Ethernet HWaddr  
02:42:f2:be:5a:21  
      inet addr:172.17.0.1 Bcast:172.17.255.255  
Mask:255.255.0.0  
          UP BROADCAST MULTICAST MTU:1500 Metric:1  
RX packets:0 errors:0 dropped:0 overruns:0 frame:0  
          TX packets:0  
errors:0 dropped:0 overruns:0 carrier:0  
          collisions:0 txqueuelen:0  
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)  
  
enp5s0    Link encap:Ethernet HWaddr  
d8:cb:8a:c2:29:4b  
      inet addr:192.168.1.13 Bcast:192.168.1.255  
Mask:255.255.255.0
```

```
        inet6 addr: fe80::8610:baf9:b53e:2cdf/64 Scope:Link
UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:10476727
errors:0 dropped:0 overruns:0 frame:0
          TX packets:9224863 errors:0
dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
RX bytes:12246353993 (12.2 GB)  TX bytes:5848871311 (5.8 GB)
Memory:fb200000-fb2fffff

lo      Link encap:Local Loopback
        inet
addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:170011 errors:0
dropped:0 overruns:0 frame:0
          TX packets:170011 errors:0 dropped:0
overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX
bytes:127632763 (127.6 MB)  TX bytes:127632763 (127.6 MB)
```

17.3 Finding adaptor info from Python

You could run a system command and parse the text to find information about IP addresses, but there's a better way:

```
pip install netifaces
```

This library will allow you to find adaptor information without having to parse text.

```
import netifaces
```

3

```
interfaces = netifaces.interfaces()
```

4

```
interfaces
```

10

```
['lo', 'enp5s0', 'br-0fc40b53eac2', 'docker0']
```

```
for iface in interfaces:
    ipaddrs = netifaces.ifaddresses(iface)
    if netifaces.AF_INET in ipaddrs:
        ipaddr_desc = ipaddrs[netifaces.AF_INET][0]
        print(f"Network interface: {iface}")
        if 'addr' in ipaddr_desc:
            print(f"\tIP address: {ipaddr_desc['addr']}")
        if 'netmask' in ipaddr_desc:
            print(f"\tNetmask: {ipaddr_desc['netmask']}")
```

9

```
Network interface: lo
    IP address: 127.0.0.1
    Netmask: 255.0.0.0
Network interface: enp5s0
    IP address: 192.168.1.13
    Netmask: 255.255.255.0
Network interface: br-0fc40b53eac2
    IP address: 172.18.0.1
    Netmask: 255.255.0.0
Network interface: docker0
    IP address: 172.17.0.1
    Netmask: 255.255.0.0
```

17.4 Finding properties of interfaces

The `ipaddress` module gives you some convenience functions for dealing with interfaces:

```
import ipaddress
```

47

```
loopback = ipaddress.IPv4Interface("127.0.0.1")
```

48

```
loopback.is_loopback
```

49

True

```
loopback.is_multicast
```

50

False

```
loopback.is_private
```

51

True

17.5 IP networks

The `ipaddress` module provides helpers for IP networks too

```
example = ipaddress.ip_network("192.168.0.0/16")
```

20

```
example.netmask
```

21

```
IPv4Address('255.255.0.0')
```

```
example.broadcast_address
```

22

```
IPv4Address('192.168.255.255')
```

```
example.network_address
```

23

```
IPv4Address('192.168.0.0')
```

```
addresses_in_subnet = example.num_addresses  
addresses_in_subnet
```

25

```
65536
```

17.6 Finding subnet address blocks

Consider the network “192.168.5.0/28”. We could figure out which addresses are on this subnet by hand, but we can also use a utility `ipaddress.ip_network`. This provides a generator which will yield out the network addresses on that subnet

```
net = ipaddress.ip_network("192.168.5.0/28")
```

27

17.7 Exercise: IP addresses on a subnet

Write a function that will take an IP address and a subnet mask length and return a list of the addresses on that subnet.

- Hint: the `ipaddress.IPv4Interface` may be convenient.

```
# See solutions/subnet_addresses.py
```

44

Once you have written a function `get_subnet_addresses()`, you should be able to use it like this:

```
address = ipaddress.IPv4Address("192.168.1.1")  
subnet_addresses = get_subnet_addresses(address, 28)  
subnet_addresses
```

45

```
[IPv4Address('192.168.1.1'),  
 IPv4Address('192.168.1.2'),  
 IPv4Address('192.168.1.3'),  
 IPv4Address('192.168.1.4'),  
 IPv4Address('192.168.1.5'),  
 IPv4Address('192.168.1.6'),  
 IPv4Address('192.168.1.7'),  
 IPv4Address('192.168.1.8'),  
 IPv4Address('192.168.1.9'),  
 IPv4Address('192.168.1.10'),  
 IPv4Address('192.168.1.11'),  
 IPv4Address('192.168.1.12'),  
 IPv4Address('192.168.1.13'),  
 IPv4Address('192.168.1.14')]
```


Chapter 18

Controlling external tools

18.1 Intro to Python's **subprocess** module

Python's `subprocess` module gives you access to calling external tools on your system. In the simplest case this means calling a command on your system and checking the output. This can be done using the `call` function.

When calling a system function with `subprocess` the process and any arguments are passed as a list of strings. This avoids the need for having to quote strings with spaces and special characters.

```
import subprocess

# Windows:
subprocess.call(['ipconfig', '/all'])
# Linux / macOS: use this instead:
# subprocess.call(['ifconfig', '-a'])
```

2

0

The response from the `call` method is the system exit code. If the process completed successfully this will be 0. Any other value indicates an error, e.g.:

```
subprocess.call(['exit', '1'], shell=True)
```

4

1

Passing the `shell=True` argument executes the command inside a shell instance, giving it access to shell environment variables and commands. This can be a security hazard, opening your system up to Shell Injection. As a general rule, do not allow your code to call a subprocess with `shell=True` where the command parameters are not set or trusted by you.

At this point you still haven't seen the output of this command. If the output of the command is important, rather than using `call`, the `check_output` command will be more useful:

```
# Windows:
print(subprocess.check_output(['ipconfig']))
# Linux / macOS:
# print(subprocess.check_output(['ifconfig']))
```

5

Windows IP Configuration

Ethernet adapter Ethernet:

```
Connection-specific DNS Suffix . :
Link-local IPv6 Address . . . . . :
    fe80::c42a:e633:dbe5:1438%2
IPv4 Address. . . . . : 10.0.2.15
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . : 10.0.2.2
```

Tunnel adapter Teredo Tunneling Pseudo-Interface:

```
Connection-specific DNS Suffix . :
IPv6 Address. . . . . :
    2001:0:9d38:6abd:1099:1929:f5ff:fd0
Link-local IPv6 Address . . . . . :
    fe80::1099:1929:f5ff:fd0%4
```

```
Default Gateway . . . . . : ::
```

Unlike `call`, `check_output` will raise an Exception when there is an error. As such it is often the preferred method of using `subprocess`.

```
subprocess.check_output(['exit', '1'], shell=True)
```

6

Exercise: Displaying the current system processes

The MS DOS `tasklist` command lists the current processes on your machine. Use `check_output` to print a list of active processes on your machine.

Extension: Add the `/?` argument to `tasklist` to see the help. Read the help and filter the tasks for `python.exe` processes.

```
#%load 4000_solution_01.py
```

7

18.2 Interacting with open processes

You've seen the simple pattern for firing a process and waiting for a response. What about the more complex pattern of communicating with a process? For this we use pipes and the `Popen` class. If you haven't heard of a pipe in reference to computer programming before, a pipe is a one way communication channel that may be used for interprocess communication. For our purposes we assume that programs we want to communicate with have three pipes:

1. The `stdin` pipe, allowing input into the process
2. The `stdout` pipe, for reading output from the process
3. The `stderr` pipe, for reading error data from the process

Under the hood, both the `call` and `check_output` functions use a `Popen` object. To simulate `check_output` you can pass in a PIPE to the `stdout` parameter:

```
proc = subprocess.Popen(['ipconfig'], stdout=subprocess.PIPE) 8
```

```
out, err = proc.communicate()  
print(out) 9
```

Windows IP Configuration

Ethernet adapter Ethernet:

```
Connection-specific DNS Suffix . :  
Link-local IPv6 Address . . . . . :  
    ↳ fe80::c42a:e633:dbe5:1438%2  
IPv4 Address. . . . . : 10.0.2.15  
Subnet Mask . . . . . : 255.255.255.0  
Default Gateway . . . . . : 10.0.2.2
```

Tunnel adapter Teredo Tunneling Pseudo-Interface:

```
Connection-specific DNS Suffix . :  
IPv6 Address. . . . . :  
    ↳ 2001:0:9d38:6abd:1099:1929:f5ff:fd0  
Link-local IPv6 Address . . . . . :  
    ↳ fe80::1099:1929:f5ff:fd0%4  
Default Gateway . . . . . : ::
```

Note that the `communicate` method will read all output from the process and wait for it to terminate. Thus you cannot communicate with a process a second time. Also note that without passing in a PIPE you cannot communicate with a process on that channel. So in the above example, even if there was an error written to the `stderr` stream, you would not see it.

Instead if you want to progressively communicate with a process, write to the input and output pipes as if you were writing to and from a file object. **WARNING** Reading from a PIPE is a blocking operation. If the process you are communicating with does not immediately respond on the stdout, then

reading may block your Python process.

```
proc = subprocess.Popen(['ipconfig'], stdin=subprocess.PIPE,  
↳   stdout=subprocess.PIPE, stderr=subprocess.PIPE)
```

10

```
print(proc.stdout.read())
```

11

Windows IP Configuration

Ethernet adapter Ethernet:

```
Connection-specific DNS Suffix . :  
Link-local IPv6 Address . . . . . :  
↳ fe80::c42a:e633:dbe5:1438%2  
IPv4 Address. . . . . : 10.0.2.15  
Subnet Mask . . . . . : 255.255.255.0  
Default Gateway . . . . . : 10.0.2.2
```

Tunnel adapter Teredo Tunneling Pseudo-Interface:

```
Connection-specific DNS Suffix . :  
IPv6 Address. . . . . :  
↳ 2001:0:9d38:6abd:1099:1929:f5ff:fd0  
Link-local IPv6 Address . . . . . :  
↳ fe80::1099:1929:f5ff:fd0%4  
Default Gateway . . . . . : ::
```

```
proc.poll()
```

12

0

Exercise: Call a subprocess with Popen

Repeat the previous exercise, replacing `check_output` with `Popen` to check the active python processes in the system.

```
#%load 4000_solution_02.py
```

13

18.3 Chaining processes together with pipes

Since the inputs and outputs of a subprocess are pipes, you can set the input pipe from another process, and use the output pipe into yet more processes. In this way you can chain together commands between disparate processes.

Consider this example: you have CSV data being read by your Python script (perhaps from a compressed dataset). You would rather this data in new-line separated JSON format. You could write your own Python script to process this data, but suppose you have the jq and csvkit tools on your computer, which are optimised to work with JSON and CSV respectively.

To solve this problem you create two processes, one calling the `csvjson` tool, and link it to another calling `jq` to process the data.

```
csvjson = subprocess.Popen(['csvjson'],
    ↵  stdin=subprocess.PIPE, stdout=subprocess.PIPE,
    ↵  stderr=subprocess.PIPE)
```

14

```
jq = subprocess.Popen(['jq', '-c', '.[]'],
    ↵  stdin=csvjson.stdout, stdout=subprocess.PIPE,
    ↵  stderr=subprocess.PIPE)
```

15

```
csvjson.stdin.write('name,id\r\nHannah,1\r\nMarcie,2\r\n')
```

16

In this case, the `csvjson` tool does not produce data until reaching the end of the file. Until we close the pipe it has not been reached.

```
csvjson.stdin.close()
csvjson.poll()
```

17

0

```
output, errors = jq.communicate()
print(output)
```

19

```
{"name": "Hannah", "id": 1}
{"name": "Marcie", "id": 2}
```

Exercise: Connecting two commands via pipes

Use the type program on Windows (or cat on Unix) to read the hamlet.txt file from the data folder. Pipe it to the clip command. Confirm this has worked by pasting the text from your clipboard.

```
#%load 4000_solution_03.py
```

27

18.4 Monitoring a log file

The Unix command tail -F follows a file. As new lines are added to the file by another process, tail updates the display. This is particularly useful for monitoring log files:

```
tail -F /var/adm/messages
```

1

This displays the last 10 lines and then blocks until more lines are added to the file, then displays additional lines too.

Here is how you can call tail -F from Python using the subprocess module:

```
import time
import subprocess
import sys

f = subprocess.Popen(['tail', '-F',
                     '/var/log/some_log_file.log'], \
```

1

```
    stdout=subprocess.PIPE, stderr=subprocess.PIPE)

while True:
    line = f.stdout.readline()
    line_text = line.decode(errors='ignore')
    sys.stdout.write(line)      # or do_something_with(line)
```

You could of course replace the final line with any other processing you like on the new line of the log file.

Note that the above solution is blocking: each call to `f.stdout.readline()` waits until a line is added to the file. A non-blocking version using `select` system call on Linux is here: <https://stackoverflow.com/a/12523371>

18.5 sh module

It is worth mentioning that the `sh` module has a simpler interface for interacting with operating system processes. Here is an example that follows a file using `tail`:

```
from sh import tail
# runs forever
for line in tail("-f", "/var/log/some_log_file.log",
    _iter=True):
    print(line)
```

1

Chapter 19

Process and system monitoring

19.1 Overview of `psutil`

The `psutil` library is designed for getting system information from your machine in a system-agnostic way, whether the operating system is Windows, macOS, Linux, or several other Unix varieties. This makes it extremely powerful especially when writing code and libraries which you are expecting to share with other users.

19.2 Examples

Examining system process times across your system tends to differ from system to system. `psutil` avoids this by allowing a common interface for this system information.

```
import psutil
```

42

For example, to get the current CPU utilization on any system you can use:

```
psutil.cpu_percent()
```

47

22.2

Likewise you can check on the available virtual memory in your system:

```
psutil.virtual_memory()
```

48

```
svmem(total=8589934592, available=2376945664, percent=72.3,
      used=7199686656, free=117219328, active=2394710016,
      inactive=2259726336, wired=2545250304)
```

This returns a named tuple containing memory information. Depending on the system you use you will end up with more or less information, but you will always have the total and available memory:

```
stat = psutil.virtual_memory()
stat.total, stat.available
```

49

```
(8589934592, 1830281216)
```

19.3 Exercise: system monitoring

Skim the docs for `psutil` here: <https://psutil.readthedocs.io>

Write a script which checks and reports whether the running system passes or fails the following health checks:

- available memory is at least 1 GB
- available disk space is at least 1 GB
- total number of incoming network packets dropped is more than 20% of number of packets received (see `net_io_counters`)
- challenge: whether all Python processes use more than 50% of total system memory. (Tip: Read about the `Process` class).
-

Chapter 20

Templating with Jinja2

Pouring text into a pre-designed mould is very useful in several contexts. One is inserting dynamically generated data (e.g. retrieved from a SQL database) into a pre-designed HTML template. Another is for creating config files for various network devices.

Recall Python’s formatted strings, or “f-strings”. These provide a simple way to create text that contains data formatted as a string:

```
name = 'Ed'  
  
f'Hello {name}'
```

1

```
'Hello Ed'
```

Jinja2 is a more powerful “templating” tool for text. It can be used to format any text file.

Jinja2 defines blocks with double braces: ‘{{ ... }}’. You can apply filters and functions to blocks - we’ll come back and see some more of those shortly. But for now this isn’t very different to a Python format string:

```
from jinja2 import Template
```

3

```
template = Template('Hello {{ name }}!')
template.render(name='John Doe')
```

```
'Hello John Doe!'
```

Jinja2 templates commonly extend over multiple lines. Best practice is to keep them in text files in a `templates` directory separate from your code-base. (In the case of HTML templates for a website, a different team may be responsible for editing them – the designers rather than the back-end developers.)

First, we create a space to store our templates:

```
! mkdir templates
```

1

```
mkdir: cannot create directory ‘templates’: File exists
```

We'll start with a simple template - a basic HTML email:

```
%%writefile templates/email.html.j2
```

2

```
<p>Hi {{ name }}</p>
```

Writing `templates/email.html.j2`

Within Jupyter or IPython console, we can render HTML nicely like this:

```
from IPython.display import HTML
```

5

```
HTML('<h1>This is a big heading</h1>')
```

```
from pathlib import Path
```

7

```
template = Path('templates/email.html.j2').read_text()
```

```
HTML(
```

```
    template.render(name='Henry')
)
```

Hi Henry

20.1 Loops

Templates support loops over data items. This makes them much more powerful than the string substitution and formatting supported by Python's strings.

Suppose you have this list of dictionaries:

```
interfaces = [ {'slotid': i, 'portid': 8000 + i,
    ↵ 'description': 'eth0', 'channelGroup': 'A'}
    for i in range(8)]
```

You could write a template to render this data as follows:

```
%%writefile templates/interfaces.cfg
{% for iface in interfaces %}
interface Ethernet{{ iface.slotid }}/{{ iface.portid }}
    description {{ iface.description }}
{% endfor %}
```

Overwriting `templates/interfaces.cfg`

```
template = Path('templates/interfaces.cfg').read_text()
print(template.render(interfaces=interfaces))
```

```
interface Ethernet0/8000
    description eth0
```

```
interface Ethernet1/8001
    description eth0
```

```
interface Ethernet2/8002
    description eth0

interface Ethernet3/8003
    description eth0

interface Ethernet4/8004
    description eth0

interface Ethernet5/8005
    description eth0

interface Ethernet6/8006
    description eth0

interface Ethernet7/8007
    description eth0
```

```
text = Path('templates/interfaces.cfg').read_text()
template = Template(text)

interfaces = [{'slotid': i, 'portid': 8000 + i,
    'description': 'eth0', 'channelGroup': 'A'}
    for i in range(8)]

print(template.render(interfaces=interfaces))
```

153

```
interface Ethernet0/8000
    description eth0

interface Ethernet1/8001
    description eth0

interface Ethernet2/8002
    description eth0

interface Ethernet3/8003
    description eth0
```

```
interface Ethernet4/8004
    description eth0

interface Ethernet5/8005
    description eth0

interface Ethernet6/8006
    description eth0

interface Ethernet7/8007
    description eth0
```

20.2 Environments

Jinja2 environments let us load and use templates in a more streamlined way. They are necessary for more advanced templating features like filters and inheritance.

The environment is the base which Jinja uses to look up templates. We create the environment with one (or more) loaders, which tell Jinja where to look for templates:

```
from jinja2 import Environment, FileSystemLoader
env = Environment(loader=FileSystemLoader('templates')) #  
    ↵ this path can be absolute or relative
```

3

Now we can load a template as follows:

```
template = env.get_template('interfaces.cfg')
```

152

When we change the template on disk we'll need to get the template from the environment again like this:

```
template = env.get_template('interfaces.cfg')
```

1

20.3 Filters

Let's consider a simple example. You might have a service that is monitoring multiple systems. Say you wanted to alter people when your machine usage got too high.

```
import psutil
```

12

```
psutil.disk_usage('/home')
```

13

```
sdiskusage(total=21462233088, used=11634122752,  
↪ free=9828110336, percent=54.2)
```

If you want to know what that is in something nicer to read, you could use the `humanize` library:

```
import humanize
```

15

```
humanize.naturalsize(21462233088)
```

```
'21.5 GB'
```

Say I wanted to use the `naturalsize` function inside my Jinja template, for any number.

```
%%writefile templates/email.html.j2
```

20

```
<p>Hi {{ name }},</p>  
<p>The total size of the disk is {{ naturalsize(total_bytes)  
↪ }}
```

Overwriting `templates/email.html.j2`

```
template = env.get_template('email.html.j2')

HTML(
    template.render(name='Ed', total_bytes=21462233088,
        ↴ naturalsize=humanize.naturalsize)
)
```

21

Hi Ed,

The total size of the disk is 21.5 GB

Exercise: A nicely structured email that reports on disk size.

Update your `email.html.j2` template file so it gets the total size available, the amount of data used, and the free space. Use the `naturalsize` function inside your template to nicely render the size. Get the values from the named tuple returned by `psutil.disk_usage`.

Update your template so that instead of multiple parameters you just pass in the named tuple, to get the same result (*Hint* inside the template you can use the standard dot notation to get attributes of an object).

```
%%writefile templates/email.html.j2

<p>Hi {{ name }},</p>
<p>The total size of the disk is {{
    ↴ naturalsize(disk_usage.total) }}</p>
<p>The used space on the disk is {{
    ↴ naturalsize(disk_usage.used) }}</p>
<p>The free space on the disk is {{
    ↴ naturalsize(disk_usage.free) }}</p>
```

22

Overwriting `templates/email.html.j2`

```
disk_usage = psutil.disk_usage('/home')
disk_usage.free
```

23

9827622912

```
template = env.get_template('email.html.j2')

HTML(
    template.render(name='Henry', disk_usage=disk_usage,
        ↴ naturalsize=humanize.naturalsize)
)
```

26

Hi Henry,

The total size of the disk is 21.5 GB

The used space on the disk is 11.6 GB

The free space on the disk is 9.8 GB

Jinja also defines filters on your data. A filter is a function that transforms the input and produces some new output. Jinja has built-in filters, but also lets you define your own really easily:

```
env.filters['naturalsize'] = humanize.naturalsize
```

28

A filter is any function – it should take *at least* one argument, and remaining arguments should have defaults.

```
%%writefile templates/email.html.j2

<p>Hi {{ name }},</p>
<p>The total size of the disk is {{ disk_usage.total |
    ↴ naturalsize }}</p>
<p>The used space on the disk is {{ disk_usage.used |
    ↴ naturalsize }}</p>
<p>The free space on the disk is {{ disk_usage.free |
    ↴ naturalsize }}</p>
```

29

Overwriting templates/email.html.j2

```
template = env.get_template('email.html.j2')

HTML(
    template.render(name='Henry', disk_usage=disk_usage)
)
```

30

Hi Henry,

The total size of the disk is 21.5 GB

The used space on the disk is 11.6 GB

The free space on the disk is 9.8 GB

```
%%writefile templates/email.html.j2

<p>Hi {{ name }},</p>
<p>The total size of the disk is {{ disk_usage.total |
    &gt; naturalsize }}</p>
<p>The used space on the disk is {{ disk_usage.used |
    &gt; naturalsize }}</p>
<p>The free space on the disk is {{ disk_usage.free |
    &gt; naturalsize }}</p>
<p>The current time is {{ now | timezone_convert }}</p>
```

29

Overwriting `templates/email.html.j2`

```
env.filters['naturalsize'] = humanize.naturalsize
```

28

Exercise: writing a basic filter for Jinja2

This is an example of using `pytz` to take a time-zone ignorant `datetime` and convert it to the time in Melbourne:

```
from datetime import datetime
```

31

```
datetime.now()
```

32

```
datetime.datetime(2020, 6, 18, 6, 28, 53, 856326)
```

```
import pytz
```

33

```
now = datetime.now()
```

34

```
pytz.utc.localize(now)
```

35

```
datetime.datetime(2020, 6, 18, 6, 30, 17, 869924,  
↳ tzinfo=<UTC>)
```

36

```
melbourne_tz = pytz.timezone('Australia/Melbourne') # or  
↳ Australia/Perth, Sydney, Canberra, Adelaide, Darwin,  
↳ Brisbane, Hobart all work
```

```
utc_now = pytz.utc.localize(now)
```

38

```
utc_now.astimezone(melbourne_tz)
```

39

```
datetime.datetime(2020, 6, 18, 16, 30, 17, 869924,  
↳ tzinfo=<DstTzInfo 'Australia/Melbourne' AEST+10:00:00  
↳ STD>)
```

1. Write a function `timezone_convert` that given a `datetime` (assumed to be UTC) returns a `datetime` in the Melbourne timezone.
2. Add this function as a new filer to your environment - call it `timezone_convert`
3. Update your template so give a value called “now” it renders that `datetime` in the Melbourne timezone

4. Render the template with a time generated from `datetime.now()`

```
str(utc_now)
```

40

```
'2020-06-18 06:30:17.869924+00:00'
```

```
def timezone_convert(dt, to_tz='Australia/Melbourne',
    ← from_tz='UTC'):
    """Localize dt as the from_tz and convert to to_tz"""
    from_tz = pytz.timezone(from_tz)
    to_tz = pytz.timezone(to_tz)
    local_dt = from_tz.localize(dt)
    return local_dt.astimezone(to_tz)
```

45

```
timezone_convert(datetime.now())
```

46

```
datetime.datetime(2020, 6, 18, 16, 51, 53, 174910,
    ← tzinfo=<DstTzInfo 'Australia/Melbourne' AEST+10:00:00
    ← STD>)
```

```
timezone_convert(datetime.now(), 'Asia/Singapore')
```

48

```
datetime.datetime(2020, 6, 18, 14, 52, 13, 786245,
    ← tzinfo=<DstTzInfo 'Asia/Singapore' +08+8:00:00 STD>)
```

```
env.filters['timezone_convert'] = timezone_convert
```

49

```
%%writefile templates/email.html.j2
```

58

```
<p>Hi {{ name }},</p>
<p>The total size of the disk is {{ disk_usage.total |
    ← naturalsize }}</p>
<p>The used space on the disk is {{ disk_usage.used |
    ← naturalsize }}</p>
```

```
<p>The free space on the disk is {{ disk_usage.free |  
    naturalsize }}</p>  
<p>The current time is {{ now | timezone_convert }}</p>
```

Overwriting `templates/email.html.j2`

```
template = env.get_template('email.html.j2')  
  
HTML(  
    template.render(name='Henry', disk_usage=disk_usage,  
        now=datetime.now())  
)
```

59

Hi Henry,

The total size of the disk is 21.5 GB

The used space on the disk is 11.6 GB

The free space on the disk is 9.8 GB

The current time is 2020-06-18 16:56:12.524517+10:00

Jinja has three kinds of “bracket sets”

- Blocks are surrounded by {{ block }}
- Flow control is surrounded by {% for %}
- Comments in your template are surrounded by {# this is a comment #}

You can change this - when you set up the environment, the block markers are keyword arguments. Try this:

```
# help(Environment)
```

60

20.4 Inheritance

Prerequisite: Environments

Exercise:

1. Create a list of dictionaries representing interfaces with the following keys:
 - slotid
 - portid
 - description
 - channelGroup
2. Render your `interfaces.cfg` template, populating the data from your dictionary.

```
# Solution: see solutions/jinja2_interfaces.py
```

16

Example output:

```
!python solutions/jinja2_interfaces.py
```

94

```
interface Ethernet0/8000
    description eth0

interface Ethernet1/8001
    description eth0

interface Ethernet2/8002
    description eth0

interface Ethernet3/8003
    description eth0
```

Example: hand-rolling YAML files via templates

This is an example of writing configuration files with custom formats. Keep in mind that good packages exist for reading and writing YAML specifically (like `pyyaml`), which would simplify writing YAML in practice.

First, read the `Data/net/rootzonedb.txt` file, which is a CSV-like file with tab separators.

```
import pandas as pd

domains = pd.read_csv('Data/net/rootzonedb.txt', sep='\t',
                     index_col=0)
```

```
domains[:3]
```

Domain	Type	Sponsoring Organisation
.abogado	generic	Top Level Domain Holdings Limited
.ac	country-code	Network Information Center (AC Domain Registry...)
.academy	generic	Half Oaks, LLC

```
data = domains['Sponsoring Organisation']
data[:5]
```

```
Domain
.abogado                      Top Level Domain Holdings
    ↵ Limited
.ac                            Network Information Center (AC Domain
    ↵ Registry...
.academy                       Half Oaks,
    ↵ LLC
.accountants                   Knob Town,
    ↵ LLC
.active                         The Active Network,
    ↵ Inc
Name: Sponsoring Organisation, dtype: object
```

Exercise:

1. Create the following YAML template to output all domains and sponsor orgs:

```
%%writefile templates/domains.yml  
{% for domain, org in data.items() %}  
    {{ domain }}: "{{ org | escape }}"
{% endfor %}
```

159

Overwriting `templates/domains.yml`

2. Write a Python script to populate the YAML from the data in `Data/net/rootzonedb.txt`.

```
%%writefile solutions/domains_and_sponsors_jinja2.py  
from jinja2 import Environment, FileSystemLoader  
  
env = Environment(loader=FileSystemLoader('templates'))  
template = env.get_template('domains.yml')  
  
import pandas as pd  
domains = pd.read_csv('~/Data/net/rootzonedb.txt', sep='\t',  
                     index_col=0)  
data = domains['Sponsoring Organisation']  
  
output = template.render(data=data)  
  
with open('new_domains.yml', mode='wt', encoding='utf8') as f:  
    f.write(output)
```

165

Overwriting `solutions/domains_and_sponsors_jinja2.py`

```
!python solutions/domains_and_sponsors_jinja2.py
```

166

```
!head -n10 new_domains.yml
```

169

```
.abogado: "Top Level Domain Holdings Limited"  
.ac: "Network Information Center (AC Domain Registry) c/o  
↳ Cable and Wireless (Ascension Island)"  
.academy: "Half Oaks, LLC"  
.accountants: "Knob Town, LLC"  
.active: "The Active Network, Inc"
```

Exercise:

- Use the `pyyaml` module (`import yaml`) to try loading the new YAML file you have written.

You'll notice that it's not valid YAML.

Challenge exercise:

Fix your YAML template so it's properly quoted.

```
def yaml_escape(thing):  
    return thing.replace('"', '\\"')
```

172

```
%%writefile templates/domains.yml  
{% for domain, org in data.items() %}  
    {{ domain }}: "{{ escape(org) }}"  
{% endfor %}
```

178

Overwriting `templates/domains.yml`

```
from jinja2 import Environment, FileSystemLoader

env = Environment(loader=FileSystemLoader('templates'))
template = env.get_template('domains.yml')

import pandas as pd
domains = pd.read_csv('~/Data/net/rootzonedb.txt', sep='\t',
    ↵ index_col=0)
data = domains['Sponsoring Organisation']

output = template.render(data=data, escape=yaml_escape)

with open('new_domains.yml', mode='wt', encoding='utf8') as
    ↵ f:
    f.write(output)
```

179

Chapter 21

Python concurrency overview

Due to limitations from physics it's been harder and harder to get more CPU performance by increasing clock speed and adding more power. As a result modern CPU design has very strongly moved in the direction of adding more CPU cores:

```
import os
cpu_count = os.cpu_count()
print(f"we have {cpu_count} CPUs, how do we use them all at
↪ once?")
```

3

we have 12 CPUs, how do we use them all at once?

On this laptop for example there's multiple CPU cores and even lower end devices, including many phone models, are starting to introduce multiple CPU cores. In the early years of Python in the early 1990's there were far fewer multi core systems and as hardware has evolved so have the concurrency options of Python.

If you have a situation where you need to run multiple computations at the same time, Python gives you the following concurrency options (including but

not limited to):

- asyncio (Since python 3.5)
- Python threads
- GIL-released threads
- multiprocessing
- distributed tasks

21.1 The “minimum schedulable unit”

The minimum schedulable unit of work is the smallest chunk of work that can be scheduled to be run.

More precisely it's individual semantic units of code that can be decided to be executed by the scheduler before control is returned to the scheduler.

This is an important concept because for concurrency we have to both define what code will run and also *when that code is run*.

21.2 Asyncio

- Built in language support since Python 3.5 (partially since Python 3.4)
- There's only one event loop and thus only one co-routine runs at any given time
- MSU: “awaitable block”
- event loop runs these awaitable blocks

Good when:

- Your problem is IO bound
- You are starting a new codebase without synchronous legacy code

Downsides:

- If code is inherently synchronous like accessing a database then asyncio can degrade performance compared to using threads

What this *doesn't* solve:

- Running code across multiple cores simultaneously, asyncio runs in one thread only.

```
import asyncio
async def print_after_sleep(to_print: str):
    await asyncio.sleep(5)
    print (to_print)

loop = asyncio.get_event_loop()

print_hello = loop.create_task(print_after_sleep("Hello"))
try:
    loop.run_until_complete(print_hello)
finally:
    print("Closing loop")
    loop.close()
```

Closing loop

Hello

21.3 Python threads

Like many other languages Python supplies the ability to create threads.

There's some issues however with the Global Interpreter Lock (commonly referred to by the acronym GIL) that substantially impacts the performance of threaded code in Python.

- One thread runs (Due to the GIL)
- Some things release the GIL however like sleeps and IO
- MSU: python bytecode
- global state is shared but is only consistent for single-bytecode operations
- combined scheduling

Good when:

- You need preemptive multitasking (be able to interrupt other threads)
- Integrating synchronous code
- Situations which require fine grained concurrency
- Python glue to other languages like C/Rust/ etc

Downsides:

- Can exhaust the virtual memory on 32 bit machines if you start a lot of threads since each thread is quite heavy on virtual memory.

What this *doesn't* solve:

- CPU bound concurrency if the GIL is not released

The GIL presents a big gotcha and performance can be substantially degraded by introducing threads if the GIL is not released. This is because the locking happens in the python interpreter itself so only one piece of bytecode can be evaluated at once. Because you can induce the thread scheduling and context switching overhead for effectively no benefit in this case performance can greatly suffer. Consider multiprocessing if you need multiple cores to be utilized simultaneously.

21.4 GIL released threads

- Multiple GIL released threads *can* run simultaneously (since there's no lock preventing this)
- MSU: host processor instruction
- global state is shared but is unreliable
- OS-scheduled

This is what allows libraries such as Numpy to be so fast, the Numpy library calls compiled code that does not have any GIL limitations.

This is mostly something that library implementers have to consider, for example FFI calls.

Python functions implemented in C, with `Py_BEGIN_ALLOW_THREADS`, will not acquire a the global interpreter lock.

Good when:

- Using 3rd party libraries like numpy that let you get the benefits of fast compiled code that will not cause the Python interpreter to lock

Downsides:

- The GIL exists to protect the internal state of the Python interpreter

itself, so if you don't acquire these locks you have less protections when things go wrong. Specifically the failure modes can involve the Python interpreter itself crashing if you get things wrong when you write this sort of code.

21.5 Multiprocessing

- multiple processes run simultaneously
- child processes are copied over completely, this however leads to situations where the global state isn't actually shared, it starts the same but can/will change over time

See:

```
import os  
os.fork
```

1

Also see `ProcessPoolExecutor` from the `concurrent.futures` library in the standard library

Good when:

- You have tasks that can be executed in parallel without needing substantial inter-task communication
- You have CPU bound tasks that benefit with the ability to have bytecode interpreted by the runtime run fully unhindered by the internal interpreter locks (GIL)

Downsides:

- Can run into issues with RAM usage due the fact that a complete copy of program state is created for each process by default. For example if your script takes 1GB of RAM then you create 8 processes you will now have $1\text{GB} \times 8 = 8\text{GB}$ of RAM usage. If RAM usage becomes a bottleneck you may have to use something like mmap backed storage to share state across processes, this comes with its own issues however.

What this doesn't solve:

- Not a great solution for IO bound latency, you'll pay a substantial memory overhead for multiple processes but won't get much of a win over asyncio in a IO bound case.

21.6 Distributed tasks

- eg Dask, Hadoop
- MSU varies, often the entire application for some subset of data
- central orchestrator

Good when:

- You have a highly segmentable and distributable workload
- The need for shared state is minimal (network latency is exceedingly bad in this case)
- Data sets are too big to handle in memory easily

Downsides:

- Running any computation over multiple machines introduces significant complexity. For example you can't rely on function calls succeeding due to issues like network timeouts. This can make error handling substantially more complex.
- Failure modes require different debugging techniques since they might not be in the same process or even same machine anymore.

Chapter 22

Threads

When you run a Python program the flow of execution of the code starts at the top of files and works it's way downwards a line at a time.

The flow of execution will change if you call a function as the code will go here to keep executing.

The model of execution is quite simple but has a major limitation, what if you want to do more than one thing at once?

For many modern operating systems to be useful they have to be able to run multiple tasks at the same time. To do this threading is provided as a way to be able to run multiple programs at once.

One of the simplest concurrency paradigms is the notion of the thread, this is supported in the Python standard library in the `threading` module
<https://docs.python.org/3/library/threading.html>

```
# Python's thread library
import threading
```

3

22.1 What is a thread

A thread is effectively a separate flow of control of execution of code. Threads do not provide a separate process for execution so there's shared state with multiple flows of execution. (Note that thread local data is provided in case a thread needs its own memory)

To do this there's 2 things that the operating system will have to provide:

1. task scheduler
2. task runner

Given that more than one operation is possible at once the operating system has to have a way of dealing with not just *how* to execute code but *when* to execute that code as well. This detail ends up being an important aspect of heavily concurrent code because the cost of switching between multiple threads is non-zero.

22.2 Creating a thread

We can create a thread using the `threading.Thread` by passing in a function to run on the thread as a target. We then have to run the thread with `.start()`.

For example:

```
from time import sleep
from threading import Thread
t = Thread(target=lambda: sleep(30))
t.start() # Start executing the target function on the thread
t.join() # Wait for Thread to finish
```

1

Exercise

- Create 1000 threads that sleep
- Look at the memory usage for your code in your operating systems
(hint: `os.getpid()` might help you find the current process more easily)

Note that you may want to not run this in Jupyter-notebook.

```
# Solution: scripts/thread_memory_usage.py
import os
from time import sleep
from threading import Thread
print(f'PID = {os.getpid()}')

NUM_THREADS = 1000
threads = [Thread(target=lambda: sleep(60)) for _ in
    range(NUM_THREADS)]
[t.start() for t in threads]
[t.join() for t in threads]
```

22.3 Finding the memory usage

top/htop are a good way to show memory usage from the command line on *nix systems such as MacOS and Linux.

top -p PID will give you info for the process with process number PID.

Advanced: the files /proc/PID/maps and /proc/PID/smaps contain more detailed memory usage.

So when I ran my script I had the PID 7170:

```
$ top -p 7170
PID USER
PR NI VIRT   RES   SHR S %CPU %MEM      TIME+ COMMAND
7170 janis    20   0 13.795g  22512   5572 S  0.0  0.3
    0:00.24 python3
```

As you can see there's a huge amount of virtual memory assigned here due to each thread having memory assigned to it. Very little of this memory is actually in active use however.

32 bit architectures note:

Note that creating a large number of threads is a substantial issue on 32 bit architectures, you simply run out of memory address space quickly if you have a one-thread-per-task architecture and have a lot of tasks to run. I have seen this be an issue on high throughput web servers in the past that were running on 32bit operating systems. If you can use an alternative approach such as `asyncio` you will not exhaust the address space.

22.4 Simple thread example: slow IO

IO operations, especially those involving a network connection are very slow, let's put those on their own thread so they don't hold up everything else.

```
# Slow
import time

messaged_processed = 0

def delayed_print(message: str, delay: int):
    """Print the given message after delaying first"""
    time.sleep(delay)
    print(message)
    global messaged_processed
    messaged_processed += 1

delayed_print("hello", 1)
delayed_print("world", 2)
print(f"We processed {messaged_processed} messages!")
```

```
hello
world
We processed 2 messages!
```

In the slow example above we got what we expected, “hello” printed out followed by “world” a second later. However this completely hold up the process, nothing can happen while we are waiting. We also have a global

variable that keeps track of how many messages the system processed, this will turn out to cause problems as we will see shortly.

22.5 Running tasks on a thread

We can run a function on a thread using the `threading.Thread` interface:

```
import time
from threading import Thread

messaged_processed = 0

def delayed_print(message: str, delay: int):
    """Print the given message after delaying first"""
    time.sleep(delay)
    print(message)
    global messaged_processed
    messaged_processed += 1

threads = [
    Thread(target=delayed_print, args=("hello", 1)),
    Thread(target=delayed_print, args=("world", 2)),
]

print(f"{len(threads)} messages to process...")
# Start our threads
[t.start() for t in threads]
# Wait for the threads to finish
[t.join() for t in threads]
print(f"Processed {messaged_processed} messages")
```

```
2 messages to process...
hello
world
Processed 2 messages
```

22.6 Which thread are we running on?

sometimes you want to know what thread you are running on:

```
threading.get_ident()
```

8

```
140159479781184
```

```
threading.current_thread().name
```

2

```
'MainThread'
```

Exercise

- Create a function that prints out a message `msg` after a specified waiting time parameter `delay`. This can take keyword arguments that modify what's printed, eg `uppercase=True` makes the message uppercase before printing it
- Run this function from a thread

```
def delayed_print(message: str, delay: int, *,  
    ↵ uppercase=False):  
    """Print the given message after delaying first"""  
    time.sleep(delay)  
    if uppercase:  
        message = message.upper()  
    print(message)  
    global messaged_processed  
    messaged_processed += 1  
  
t = Thread(target=delayed_print, args=("hello world", 1),  
    ↵ kwargs={"uppercase":True})  
t.start()  
t.join()
```

15

```
HELLO WORLD
```

22.7 Locks

When you have a single thread of execution of code you don't have to worry about 2 separate cores of execution accessing the same piece of data at once, the bugs that come up because due to ordering of execution are known as race conditions and are some of the most nasty bugs that software engineers will face.

Acquiring a lock is a way in which you can make sure that only one thread is going to access a variable at the same time. Note that a lock *prevents* other threads of execution from also accessing the same variable.

If you can provably make your program work in parallel without locks it is highly advantageous to not use locks. Making programs that will be *provably* correct in a concurrent setting without using locks is hard, which is why for example the CPython interpreter still extensively uses a global interpreter lock to make sure internal consistency is maintained. You may wish to look at Non-blocking algorithms and lock free programming techniques, though this is beyond the scope of this current document.

```
import time
from threading import Thread

messaged_processed = 0

def delayed_message_processing(message: str, delay: int):
    time.sleep(delay)
    messaged_processed = messaged_processed + 1

threads = []
for _ in range(1000):
    threads.append(Thread(target=delayed_message_processing,
                          args=("test", 1)))

print(f"{len(threads)} messages to process...")
# Start our threads
[t.start() for t in threads]
# Wait for the threads to finish
```

```
[t.join() for t in threads]
print(f"Processed {messaged_processed} messages")
```

1000 messages to process...

Deadlocking

If A is locked and is waiting on B and B is locked waiting on A we have a condition known as a deadlock. Because both threads are simultaneously waiting on each other no progress can be made. Try to avoid such situations in your code. Here's an example from the `concurrent.futures` documentation:

```
from concurrent.futures import ThreadPoolExecutor
import time

def wait_on_b():
    time.sleep(5)
    print(b.result()) # b will never complete because it is
                      # waiting on a.
    return 5

def wait_on_a():
    time.sleep(5)
    print(a.result()) # a will never complete because it is
                      # waiting on b.
    return 6

executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(wait_on_b)
b = executor.submit(wait_on_a)
print(a.running())
print(b.running())
time.sleep(10)
print(a.running())
print(b.running())
```

17

```
True  
True  
True  
True
```

22.8 Thread pools

Creating and destroying threads can be much more expensive than whatever it is you are actually trying to do. Since there's a maximum number of threads your processor can actually process at any given time it makes sense to not create a lot more threads than whatever number that is. Creating a pool of threads can substantially improve performance (but not always see GIL example below for a case where more threads means worse performance)

22.9 The global interpreter lock (GIL)

If more than one thread of execution could access internal Python data structures some very nasty bugs could occur in the interpreter itself which could lead to interpreter crashing, or worse, incorrect computations from occurring. So for the interpreter to protect its internal state it acquires a global lock, this is very often referred to in the community by the acronym GIL.

Quick summary:

- The GIL prevents the CPython interpreter from executing more than one piece of bytecode at the same time
- Actions like `time.sleep()` don't run bytecode and hence do not hold this lock
- IO also does not hold the interpreter lock
- You will notice that in your operating systems task manager CPython programs tend to use only one core unless explicitly designed otherwise (see our other materials for how you can do this in your code)

Threading will allow you to get concurrency in cases where bytecode is *not* executing. So for most operations that wait on IO you can get better processor utilization and less latency by letting the waiting occur in threads. (Note that for IO tasks the Asyncio library and approach may be substantially easier to

manage)

There's many articles about the GIL, but the one of the main issues/gotcha's is that if you need parallel computation with multiple bytecode operations being executed simultaneously if the GIL is acquired it will prevent you from being able to achieve this.

Running parallel computations in threads with the GIL being acquired will decrease performance compared to not using threads at all. This is because you will incur thread scheduling overhead without any additional instructions being able to be executed, the more threads you add when the GIL is the bottleneck the worse your performance will get.

```
import math
import os
def slow_example(run_number):
    """Run an expensive function that doesn't release the
    ↵ GIL"""
    print(run_number)
    print("Executing our Task on Process:
        ↵ {}".format(os.getpid()))
    y = [1]*10000000
    [math.exp(i) for i in y]
```

29

```
%%timeit
NUM_TRIALS = 4

print("Start")
list(map(slow_example, list(range(NUM_TRIALS))))
print("End")
```

30

```
Start
0
Executing our Task on Process: 9938
1
Executing our Task on Process: 9938
2
```

```
Executing our Task on Process: 9938
3
Executing our Task on Process: 9938
End
Start
0
Executing our Task on Process: 9938
1
Executing our Task on Process: 9938
2
Executing our Task on Process: 9938
3
Executing our Task on Process: 9938
End
Start
0
Executing our Task on Process: 9938
1
Executing our Task on Process: 9938
2
Executing our Task on Process: 9938
3
Executing our Task on Process: 9938
End
Start
0
Executing our Task on Process: 9938
1
Executing our Task on Process: 9938
2
Executing our Task on Process: 9938
3
Executing our Task on Process: 9938
End
Start
0
Executing our Task on Process: 9938
1
Executing our Task on Process: 9938
```

```
2
Executing our Task on Process: 9938
3
Executing our Task on Process: 9938
End
Start
0
Executing our Task on Process: 9938
1
Executing our Task on Process: 9938
2
Executing our Task on Process: 9938
3
Executing our Task on Process: 9938
End
Start
0
Executing our Task on Process: 9938
1
Executing our Task on Process: 9938
2
Executing our Task on Process: 9938
3
Executing our Task on Process: 9938
End
Start
0
Executing our Task on Process: 9938
1
Executing our Task on Process: 9938
2
Executing our Task on Process: 9938
3
Executing our Task on Process: 9938
End
4.64 s ± 357 ms per loop (mean ± std. dev. of 7 runs, 1 loop
↳ each)
```

```
%%timeit
import math
from concurrent.futures import ThreadPoolExecutor
NUM_TRIALS = 4

print("Start")
with ThreadPoolExecutor(max_workers=4) as executor:
    result = executor.map(slow_example, range(NUM_TRIALS))
print("End")
```

```
Start
0
Executing our Task on Process: 9938
1
Executing our Task on Process: 9938
2
Executing our Task on Process: 9938
3
Executing our Task on Process: 9938
End
Start
0
Executing our Task on Process: 9938
1
Executing our Task on Process: 9938
2
Executing our Task on Process: 9938
3
Executing our Task on Process: 9938
End
Start
0
Executing our Task on Process: 9938
1
Executing our Task on Process: 9938
2
Executing our Task on Process: 9938
```

```
3
Executing our Task on Process: 9938
End
Start
0
Executing our Task on Process: 9938
1
Executing our Task on Process: 9938
2
Executing our Task on Process: 9938
3
Executing our Task on Process: 9938
End
Start
0
Executing our Task on Process: 9938
1
Executing our Task on Process: 9938
2
Executing our Task on Process: 9938
3
Executing our Task on Process: 9938
End
Start
0
Executing our Task on Process: 9938
1
Executing our Task on Process: 9938
2
Executing our Task on Process: 9938
3
Executing our Task on Process: 9938
End
Start
0
Executing our Task on Process: 9938
1
Executing our Task on Process: 9938
2
```

```
Executing our Task on Process: 9938
3
Executing our Task on Process: 9938
End
Start
0
Executing our Task on Process: 9938
1
Executing our Task on Process: 9938
2
Executing our Task on Process: 9938
3
Executing our Task on Process: 9938
End
6.44 s ± 204 ms per loop (mean ± std. dev. of 7 runs, 1 loop
↳ each)
```

22.10 What to do instead?

Notice how adding the thread pool there made things slower not faster! With more threads running this just gets worse (try it out). Since the `ProcessPoolExecutor` in `concurrent.futures` shares the same API as the `ThreadPoolExecutor` we can use this as a drop in replacement for the sake of this example.

```
%%timeit
import math
from concurrent.futures import ProcessPoolExecutor
NUM_TRIALS = 4

print("Start")
with ProcessPoolExecutor(max_workers=4) as executor:
    result = executor.map(slow_example, range(NUM_TRIALS))
print("End")
```

35

```
Start
0
```

```
1
2
3
Executing our Task on Process: 12023
Executing our Task on Process: 12025
Executing our Task on Process: 12022
Executing our Task on Process: 12024
End
Start
2
1
3
0
Executing our Task on Process: 12047
Executing our Task on Process: 12046
Executing our Task on Process: 12048
Executing our Task on Process: 12045
End
Start
1
0
3
2
Executing our Task on Process: 12068
Executing our Task on Process: 12067
Executing our Task on Process: 12069
Executing our Task on Process: 12070
End
Start
0
1
3
2
Executing our Task on Process: 12090
Executing our Task on Process: 12091
Executing our Task on Process: 12093
Executing our Task on Process: 12092
End
Start
```

```
1
3
0
2
Executing our Task on Process: 12113
Executing our Task on Process: 12114
Executing our Task on Process: 12112
Executing our Task on Process: 12115
End
Start
1
3
0
2
Executing our Task on Process: 12134
Executing our Task on Process: 12135
Executing our Task on Process: 12137
Executing our Task on Process: 12136
End
Start
1
0
3
Executing our Task on Process: 12158
2
Executing our Task on Process: 12157
Executing our Task on Process: 12156
Executing our Task on Process: 12159
End
Start
0
2
1
3
Executing our Task on Process: 12180
Executing our Task on Process: 12182
Executing our Task on Process: 12183
Executing our Task on Process: 12181
End
```

1.72 s ± 98.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop
↳ each)

Chapter 23

Concurrency with concurrent.futures

If you write a lot of concurrent code you'll notice that you end up having to do various types of tasks over and over again, including:

- Checking if a computation is finished
- Finding out the result of the computation
- Checking if a task is running
- Cancelling a task
- Dealing with timeout conditions

Because all these things are so common there was a proposal to make an interface in the Python standard library: <https://www.python.org/dev/peps/pep-3148/>

This interface is found in `concurrent.futures` since Python 3.2.

23.1 Backported version

If you need to get a backported version for an older version you will have to first install the backported version with:

```
pip install futures
```

Note however that if you are running Python 2.7 this will not support `ProcessPoolExecutor` because of the following open issue <https://github.com/agronholm/pythonfutures/issues/29> and <https://bugs.python.org/issue9205> (If you need this please consider porting your code to a more recent Python 3 version)

If you are attempting to support both Python 2 and 3 from the same code base you will have to make sure that `futures` is only installed on Python 2.7, you can do this in your `setup.py` as follows:

```
setup(  
    ...  
    install_requires=[  
        'futures; python_version == "2.7"'  
    ]  
)
```

1

23.2 Scheduling a task to run concurrently

When you deal with any form of concurrency you have to not only specify what code will run, you also have to specify when and how that code will run. `concurrent.futures` gives us an abstract `Executor` interface which will allow us to schedule tasks to run. Concrete implementations of this interface will specify how these tasks will be executed. We will use a `ThreadPoolExecutor` for many of these examples.

```
import concurrent.futures  
import threading  
import time  
  
def task(time_to_wait: int):  
    """A task that will take `time_to_wait` seconds to  
    ↴ complete"""  
    print(f"Currently running on thread  
    ↴ {threading.current_thread().name}")  
    time.sleep(time_to_wait)
```

21

```
print(f"Task completed on thread\n"
      f"↳ {threading.current_thread().name}")
return time_to_wait

exe = concurrent.futures.ThreadPoolExecutor(max_workers=2)
print("Main thread")
fut = exe.submit(task, 3)
print(type(fut))
print(fut.done())
time.sleep(5)
print(fut.done())
print("Result:", fut.result())
exe.shutdown(wait=True) # good to clean up properly
```

```
Main thread
Currently running on thread ThreadPoolExecutor-5_0
<class 'concurrent.futures._base.Future'>
False
Task completed on thread ThreadPoolExecutor-5_0
True
Result: 3
```

23.3 The Future class

When we submit a task to run in an Executor we get a `Future` result returned from the executor. This allows us to deal with the asynchronous nature of the task.

There's a few parts involved in using such an object.

Since a `Future` object is running asynchronously we will need to check if the task is completed before we can use the results.

- `fut.done()` will return if the task has completed or not.
- `fut.cancel()` will attempt to cancel the task, this can only be done before the execution of the task has commenced.
- `fut.cancelled()` will return if the task was cancelled.
- `fut.result()` will wait indefinitely for the task to complete, NOTE

this will block

If we want to wait a shorter amount of time for the tasks to complete then we should specify the `timeout` argument for the `result()` method call so we don't wait forever.

23.4 A motivating example: a group of slow tasks

Some tasks take a lot of time to execute but don't necessarily involve a lot of resources. One good example is Network latency bound tasks another is waiting on a remote end to perform a task.

A good example of this is looking up a slow API endpoint, in many situations you don't want to have your codes execution be blocked while waiting on a slow network API call. A good example of an API endpoint that takes a while to return data is from the HTTP requests testing server (httpbin). In particular we will make use of the endpoint `/delay` that's only purpose is to delay for the given number of seconds on the server before it returns us the data from the request.

```
import requests  
slow_url_1 = "https://httpbin.org/delay/1"  
slow_url_5 = "https://httpbin.org/delay/5"
```

7

```
response = requests.get(slow_url_1)  
response.elapsed
```

8

```
datetime.timedelta(seconds=2, microseconds=88922)
```

```
response = requests.get(slow_url_5)  
response.elapsed
```

9

```
datetime.timedelta(seconds=6, microseconds=92957)
```

```
response.json()
```

11

```
{'args': {},  
 'data': '',  
 'files': {},  
 'form': {},  
 'headers': {'Accept': '*/*',  
             'Accept-Encoding': 'gzip, deflate',  
             'Host': 'httpbin.org',  
             'User-Agent': 'python-requests/2.19.1'},  
 'origin': '123.243.137.65, 123.243.137.65',  
 'url': 'https://httpbin.org/delay/5'}
```

23.5 API calls: The slow synchronous way

We could call these endpoints then store the results from these calls synchronously, but we will end up having to wait a while:

```
%%timeit  
  
import requests  
URLS = [  
    "https://httpbin.org/delay/1",  
    "https://httpbin.org/delay/3",  
    "https://httpbin.org/delay/5",  
]  
  
def get_results(url):  
    """Fetch results from URL via a GET request"""  
    response = requests.get(URL)  
    return response.json()  
  
results = []  
for URL in URLS:  
    results[URL] = get_results(URL)
```

35

12.3 s ± 39.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop
↳ each)

results

36

```
{'https://httpbin.org/delay/1': {'args': {},
  'data': '',
  'files': {},
  'form': {},
  'headers': {'Accept': '*/*',
    'Accept-Encoding': 'gzip, deflate',
    'Host': 'httpbin.org',
    'User-Agent': 'python-requests/2.19.1'},
  'origin': '123.243.137.65, 123.243.137.65',
  'url': 'https://httpbin.org/delay/1'},
 'https://httpbin.org/delay/3': {'args': {},
  'data': '',
  'files': {},
  'form': {},
  'headers': {'Accept': '*/*',
    'Accept-Encoding': 'gzip, deflate',
    'Host': 'httpbin.org',
    'User-Agent': 'python-requests/2.19.1'},
  'origin': '123.243.137.65, 123.243.137.65',
  'url': 'https://httpbin.org/delay/3'},
 'https://httpbin.org/delay/5': {'args': {},
  'data': '',
  'files': {},
  'form': {},
  'headers': {'Accept': '*/*',
    'Accept-Encoding': 'gzip, deflate',
    'Host': 'httpbin.org',
    'User-Agent': 'python-requests/2.19.1'},
  'origin': '123.243.137.65, 123.243.137.65',
  'url': 'https://httpbin.org/delay/5'}}}
```

23.6 Managing the flow of control: callbacks

As you can see with those requests it takes a long time for us to get any results due to the combination of network latency and the time we have to wait while the server is “processing” our request. If you have a large number of requests you’ll end up potentially waiting a *long* wall-clock-time. If we are to run these tasks in parallel then we could see a lot of speed up since calling the API does not rely on any other previous call occurring in any order. This does introduce an additional complexity, because any API call can complete in any order we need some way of compiling the results, the simple loop from before will not be able to do this.

Perhaps the simplest way to deal with this is to use a callback function, the nature of such a function is that when the task is done it “calls back” to the function we provided. In that callback function we can then process the results as we wish. `fut.add_done_callback` will allow us to register an existing function as a callback function on a task completion.

When we specify a function as a callback it receives the future that called it as a parameter. If we need to do something when the task is completed, such as store results, that requires the parameters used when calling the future in the first place we will have to keep track of them. Futures can be stored as dictionary keys so using them in a dictionary as the key with the information that is needed to be stored for later is a good approach. That’s the purpose of `jobs` in the following example:

```
%%timeit
import requests
URLS = [
    "https://httpbin.org/delay/1",
    "https://httpbin.org/delay/3",
    "https://httpbin.org/delay/5",
]
results = {}

def get_results(url):
    """Fetch results from URL via a GET request"""
    response = requests.get(url)
    results[url] = response.json()
    print(f"Completed {url} at {response.elapsed.total_seconds()}")
    fut.add_done_callback(lambda f: print(f"Completed {f.result().url} at {f.result().elapsed.total_seconds()}"))
```

37

```
response = requests.get(URL)
return response.json()

def url_done(fut):
    """Callback when the future is done"""
    url = jobs[fut]
    results[url] = fut.result()

exe = concurrent.futures.ThreadPoolExecutor(max_workers=5)

jobs = {}
for URL in URLs:
    fut = exe.submit(get_results, URL)
    jobs[fut] = URL
    fut.add_done_callback(url_done)
exe.shutdown(wait=True) # good to clean up properly
```

6.11 s ± 8.87 ms per loop (mean ± std. dev. of 7 runs, 1 loop
↳ each)

results

34

```
{'https://httpbin.org/delay/1': {'args': {},
  'data': '',
  'files': {},
  'form': {},
  'headers': {'Accept': '*/*',
  'Accept-Encoding': 'gzip, deflate',
  'Host': 'httpbin.org',
  'User-Agent': 'python-requests/2.19.1'},
  'origin': '123.243.137.65, 123.243.137.65',
  'url': 'https://httpbin.org/delay/1'},
'https://httpbin.org/delay/3': {'args': {},
  'data': '',
  'files': {},
  'form': {},
  'headers': {'Accept': '*/*',
```

```
'Accept-Encoding': 'gzip, deflate',
'Host': 'httpbin.org',
'User-Agent': 'python-requests/2.19.1'},
'origin': '123.243.137.65, 123.243.137.65',
'url': 'https://httpbin.org/delay/3'},
'https://httpbin.org/delay/5': {'args': {},
'data': '',
'files': {},
'form': {},
'headers': {'Accept': '*/*',
'Accept-Encoding': 'gzip, deflate',
'Host': 'httpbin.org',
'User-Agent': 'python-requests/2.19.1'},
'origin': '123.243.137.65, 123.243.137.65',
'url': 'https://httpbin.org/delay/5'}}
```

23.7 Threadpool context manager

One thing you may find useful is to use the Context manager for the Executor, within the block it defines you will be able to submit jobs to it.

This context manager will call `executor.shutdown(wait=True)` when the block leaves.

```
import time
import threading

def slow_square(number, delay):
    """Simulate a slow calculation"""
    print(f"Calculating the square of {number} after {delay}
          seconds",
          f"on thread {threading.current_thread().name}")
    time.sleep(delay)
    return number ** 2
```

23

```
from concurrent.futures import ThreadPoolExecutor
with ThreadPoolExecutor(max_workers=5) as executor:
    s1 = executor.submit(slow_square, 1, 2)
    s2 = executor.submit(slow_square, 3, 2)
    s3 = executor.submit(slow_square, 4, 8)

print("Results:", s1.result(), s2.result(), s3.result())
```

Calculating the square of 1 after 2 seconds on thread

↳ ThreadPoolExecutor-6_0

Calculating the square of 3 after 2 seconds on thread

↳ ThreadPoolExecutor-6_1

Calculating the square of 4 after 8 seconds on thread

↳ ThreadPoolExecutor-6_2

Results: 1 9 16

23.8 Map

`executor.map` will calculate multiple results much like `map` but across multiple threads.

Note that there's one annoyance when you have to pass multiple arguments to a function, due to the limited API this requires a small workaround, since for `map` to be able to differentiate between the following set of calls:

1. `arg1 = [(1, 2), (3, 2), (4, 8)]`
2. `arg1 = (1, 2); arg2 = (3, 2); arg3 = (4, 8)`

We use `*zip(*args)` to specify that we wanted the second option with 3 function calls.

```
import time
from concurrent.futures import ThreadPoolExecutor
args = [(1, 2), (3, 2), (4, 8)]

with ThreadPoolExecutor(max_workers=5) as executor:
    results = executor.map(slow_square, args) # BUG!
```

```
for result in results:  
    print(result)
```

```
import time  
from concurrent.futures import ThreadPoolExecutor  
args = [(1, 2), (3, 2), (4, 8)]  
  
with ThreadPoolExecutor(max_workers=5) as executor:  
    results = executor.map(slow_square, *zip(*args))  
  
for result in results:  
    print(result)
```

1
9
16

23.9 Exercise: weather exercise with a thread pool

Earlier in this course we showed you how you could look up the current weather in a variety of cities, this is something that has a very substantial amount of network latency.

Since this is a highly IO bound task we have a good opportunity to use concurrency to speed things up (see Concurrency overview to see other options).

Note: you may use a rate-limited API, if this is the case you will want to make sure you don't make too many calls at once. OpenWeatherMap is limited to 60 API calls per minute on the free plan so make sure you don't have too long a list of cities when running concurrently.

- Use a `ThreadPoolExecutor` to send queries to the OpenWeatherMap API in parallel
- Extension: Handle timeout error cases

- Extension: Gracefully handle cancelling the remaining tasks if there's a `KeyboardInterrupt`

```
# see solutions/threadpool_weather.py
```

1

Chapter 24

Parallel processing with Dask

Dask is a new library for parallel computing and multi-core execution on datasets that are too large for memory.

NumPy's `ndarray` and Pandas' `DataFrame` are foundational data structures. These are very effective as standards because libraries can be written that don't know about each other and still interoperate. But they are mostly limited to memory and a single core. Dask offers interfaces to these tools which use blocked algorithms and dynamic task scheduling to achieve sensible parallelism.

In contrast to Hadoop and Spark (based on the JVM), Dask is simple (lightweight) and interfaces very well with the existing ecosystem of Python libraries.

```
import dask
```

1

24.1 Architecture

Dask mimics the high-level API of the `ndarray` and `DataFrame` objects, while creating task graphs under the hood with dependencies and schedules to execute computational steps in parallel. By doing so it allows you to work easily with datasets in the range of 10 GB to 100 GB on a single machine.

24.2 `dask.array`

`dask.array` is an out-of-core, multi-core, multidimensional array library. It copies the NumPy array interface using blocked algorithms. A large collection of blocked algorithms have been devised by the linear algebra community. By using it, dask is able to achieve improved CPU cache performance even in the case where all data fits in memory. It provides memory-aware dynamic scheduling.

Constructs supported by `dask.array`

- Arithmetic: `+, -, *, /, **, exp, log, sin, cos, ...`
- Reductions: `sum(), mean(), max(), std(), prod()`
- Slicing: `x[:5, 100:50:-10]`
- Fancy indexing: `x[[5, -1], :]`
- Some linear algebra: `dot, qr, svd, tensordot, ...`
- Axis reordering: `transpose(), ...`
- Parallel algorithms (`topk, ...`)
- Slightly overlapping arrays
- Integration with HDF5

Example: estimating π

Recall that a simple Monte Carlo estimator for π can be calculated using NumPy like this:

```
import numpy as np
```

2

```
def estimate_pi_numpy(N=10**7):  
    x, y = np.random.random((2, N))  
    return 4.0 * (x**2 + y**2 < 1).mean()
```

3

```
estimate_pi_numpy()
```

4

3.1413772

```
%timeit estimate_pi_numpy()
```

5

420 ms ± 16.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop
↳ each)

Now consider the same task being performed in parallel using dask arrays:

```
import dask.array as da
```

6

```
def estimate_pi_dask(N=10**7, chunks=(2, 10**6)):  
    x, y = da.random.random((2, N), chunks=chunks)  
    return 4.0 * (x**2 + y**2 < 1).mean()
```

7

```
pi = estimate_pi_dask()  
pi
```

8

dask.array<mul, shape=(), dtype=float64, chunksizer>

```
pi.compute()
```

9

3.1426784

```
%timeit estimate_pi_dask(chunks=(1, 10**6)).compute()
```

10

216 ms ± 17 ms per loop (mean ± std. dev. of 7 runs, 1 loop
↳ each)

So straight away we have a **factor-2 speedup** (on my laptop).

However, notice that the performance depends critically on a well-chosen chunks argument:

```
%timeit estimate_pi_dask(chunks=(1, 10**5)).compute()
```

11

307 ms ± 38.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop
↳ each)

```
%timeit estimate_pi_dask(chunks=(1, 10**7)).compute()
```

12

282 ms ± 13.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop
↳ each)

```
%timeit estimate_pi_dask(chunks=(1, 10**4)).compute()
```

13

1.66 s ± 53.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop
↳ each)

Performance comparison with `ipyparallel`

We can compare the performance of the code above to code using `ipyparallel`:

Run in terminal:

```
ipcluster start -n 8
```

1

```
from ipyparallel import Client
```

32

```
rc = Client()
```

33

```
rc.ids
```

34

```
[0, 1, 2, 3, 4, 5, 6, 7]
```

```
%px import numpy as np
```

35

```
rc[:].map_sync(estimate_pi_numpy, [10**6] * 10)
```

36

```
[3.1404960000000002,
 3.1432440000000001,
 3.1428199999999999,
 3.143132,
 3.1416599999999999,
 3.1417760000000001,
 3.141432,
 3.1416599999999999,
 3.141864,
 3.1422880000000002]
```

```
%timeit np.mean(rc[:].map_sync(estimate_pi_numpy, [10**6] *
                                10))
```

37

```
1 loop, best of 3: 247 ms per loop
```

Starting with 4 engines instead of 8 makes no difference to the timing (on my laptop).

A side note on visualisation

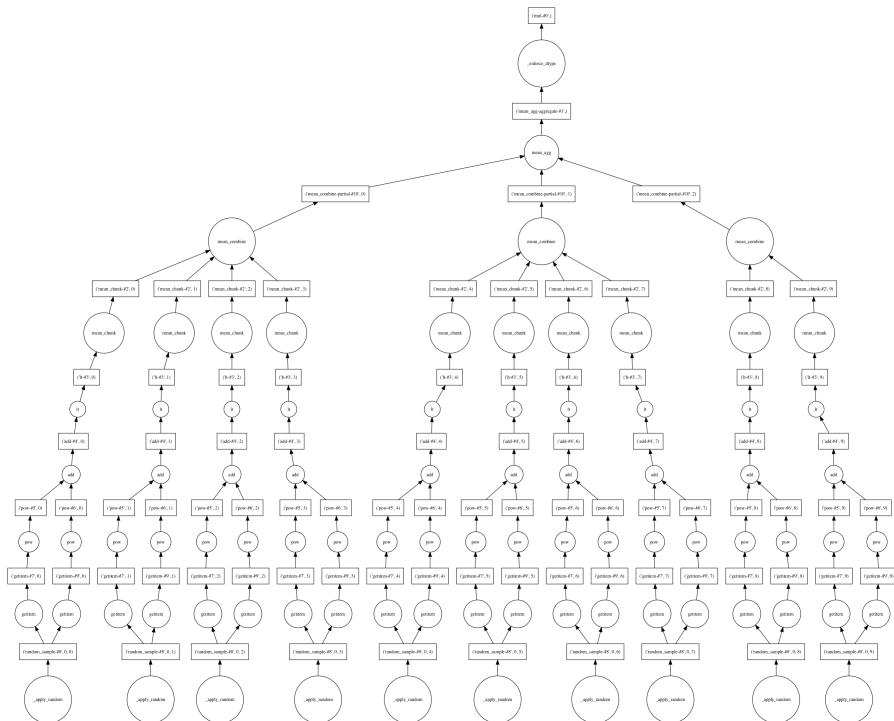
Dask strings together operations as a directed graph. Visualising the graph can be a great way of finding out how your tasks are being spread out across

multiple processes (not always useful as the processes get larger and more complex, but still fun!).

Dask does not do the visualisation in Python - instead it uses the excellent dot command line utility for generating directed graph trees from the graphviz (<https://www.graphviz.org/>) toolkit. You must have this tool installed (and available on your system PATH environment variable) and the python graphviz library (which can be installed with `pip install graphviz`) if you want to generate the visualisation.

```
pi.visualize()
```

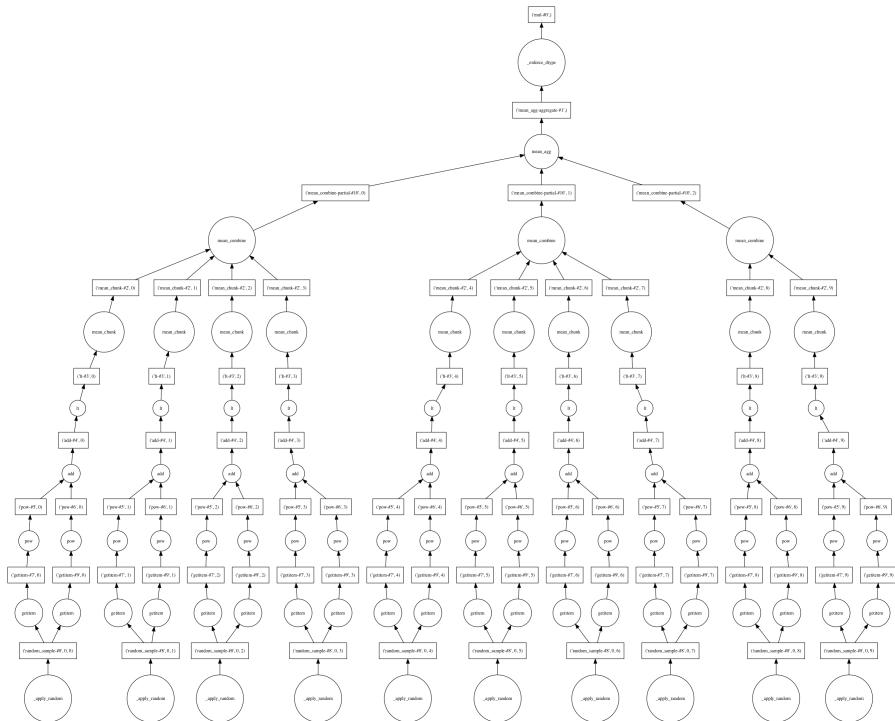
14



By default when visualising a task graph in Jupyter the `visualize` method will automatically generate a temporary PNG file and load it in the page. You can also supply a filename with an extension to generate the file and save it to disk:

```
pi.visualize('pi_estimate.png')
```

15



```
pi.visualize('pi_estimate.pdf')
```

16

```
!open pi_estimate.pdf
```

17

Exercise: Yahtzee! as a Monte Carlo simulation

The game of Yahtzee has been around in one form or another since the 1940s. The game is played by rolling 5 dice (with 2 re-rolls) to score points in different categories. To score the titular Yahtzee you must roll the same value on each of the five dice (e.g. 5 * 6 scores). The probability of this on the first roll is easy to work out:

$$6_{numbers} * \frac{1}{6^5_{possibilities}} \approx 0.00077$$

You can generate a random roll in NumPy to simulate a roll easily:

```
roll = np.random.randint(1, 7, 5)
np.all(roll == roll[0]) # every value from the roll is the
↪ same as the first - ie a Yahtzee.
```

18

`False`

- Use dask to simulate this on a large scale with one million rolls and a chunk size of 100,000. Visualize the task graph (if you have the required libraries installed) and compute the result.

```
# %load solutions/yahtzee.py
```

19

24.3 `dask.dataframe`

The `dask.dataframe` is a large parallel dataframe composed of many smaller Pandas dataframes, split along the index.

Similarly with the `dask.array` this is an implementation of the Pandas library, including the `DataFrame` object to allow for chunked parallel processing. Again, like the `dask.array` not every function is available, but a large subset are. *Note:* one obvious missing function is the `median` function, which is actually very hard to implement through parallel computation. Instead you can use an approximate quantile, with the `.quantile(0.5)` function giving an approximate result for 50% of the data.

There are some use cases where the `dask.dataframe` is well suited to use: typically these are operations which rely on a pre-existing index. Setting the index itself, or operations which require setting an index (e.g. `groupby` on a column which is not the index) can be slower. By default the `dask.dataframe` uses the threaded scheduler, but because not all

Pandas operations release the global interpreter lock (GIL) the speedups may not be as significant as the speedups from `dask.array`.

```
import dask.dataframe as dd
```

20

Depending on the read method used there are various ways for determining the size of the data chunk. CSV for example uses a `blocksize` parameter which represents the number of bytes to be read into each block, while converting a dataframe from pandas or reading a HDF file use a `chunksize` parameter which is the number of rows per partition.

```
crashes = dd.read_hdf('Data/traffic/crashes.h5',
    ↵ key='crashes', chunkszie=10000, sorted_index=True)
```

21

For example, given a dataset with a sorted datetime index you can calculate the total number of people involved in crashes each month over a five year period:

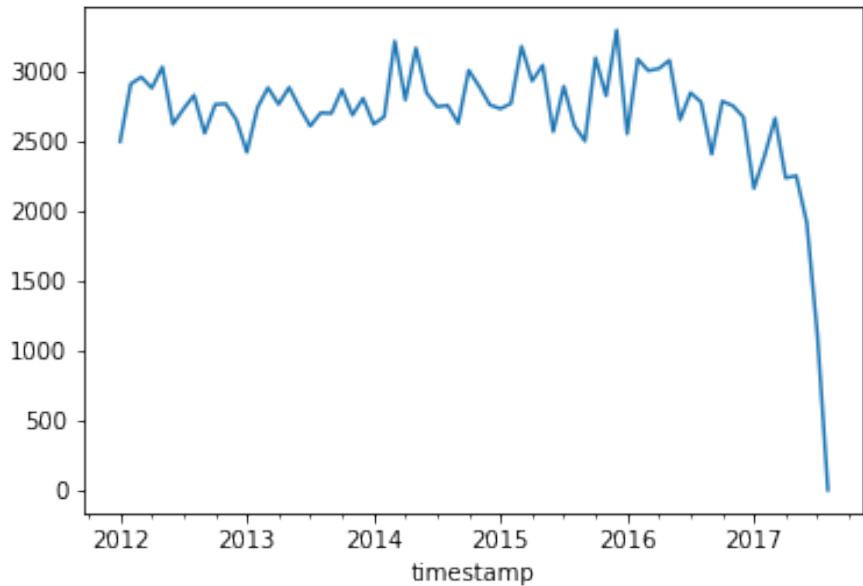
```
%matplotlib inline
```

23

```
total_persons = crashes['TOTAL_PERSONS'].resample('M').sum()
total_persons.compute().plot()
```

24

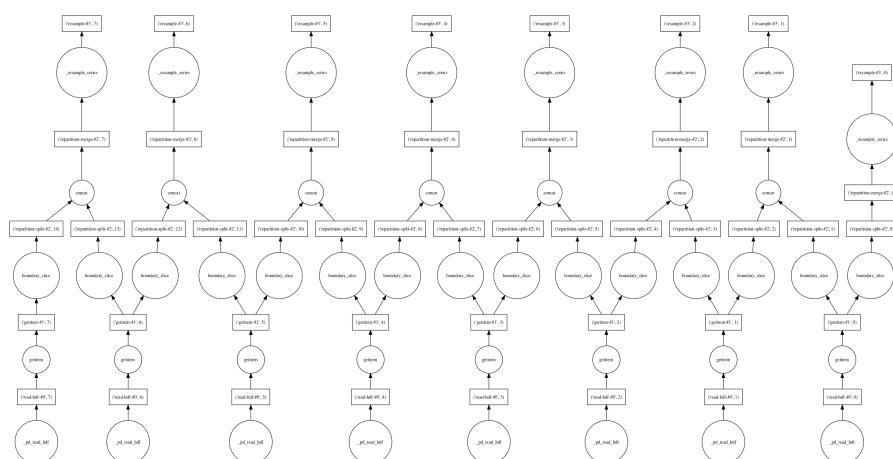
```
<matplotlib.axes._subplots.AxesSubplot at 0x123093710>
```



And like the `dask.array` instead of computing the results you can visualize the calculation graph:

```
total_persons.visualize()
```

25



Exercise: Look at a rolling window of averages per month

You'll notice on the plot above there is a significant dip when looking at the final month. This is due to there only being one sample in August (the 1st). Instead of simply aggregating, try computing the rolling 30 day average for accident numbers and plotting the result.

```
# %load solutions/crash_means.py
```

26

In this case this dataset is easily small enough to fit in memory without having to consider splitting the calculations to increase efficiency. In this case the calculations is much less efficient than calculating the results using an in-memory pandas solution.

Instead, consider the case where you have data which is potentially very large spread out across multiple files on disk:

```
import glob
glob.glob('Data/cites/*.csv')[:10]
```

27

```
Data/cites/report_1975.csv Data/cites/report_1997.csv
Data/cites/report_1976.csv Data/cites/report_1998.csv
Data/cites/report_1977.csv Data/cites/report_1999.csv
Data/cites/report_1978.csv Data/cites/report_2000.csv
Data/cites/report_1979.csv Data/cites/report_2001.csv
```

These datasets represent reported information to the UN Convention on Trade in Endangered Species of Wild Flora and Fauna (CITES) that describe the legal transport of one endangered species from one country to another. Dask allows the creation of a `dask.dataframe` from multiple files matching a `glob` pattern:

```
cites_reports = dd.read_csv('Data/cites/report_*.csv')
```

28

Note that when reading these files dask will automatically partition if you do not supply a `blocksize` based on the number of files being read from

the disk. You can even visualize resolving the multiple files into a single dataframe:

```
cites_reports.visualize()
```

29



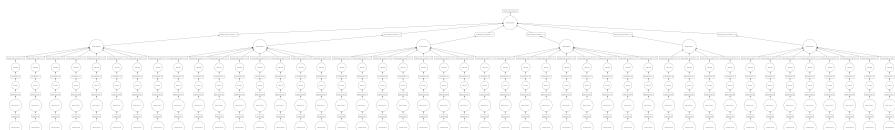
You can now determine how many endangered species Australia (country code 'AU') imported for the purpose of displaying in a Zoo (purpose 'Z') with a groupby across multiple files:

```
cites_grouper = cites_reports.groupby(['Importer',
                                       'Purpose'])
animals_transported = cites_grouper[['Importer reported
                                       quantity', 'Exporter reported quantity']].sum()
```

30

```
animals_transported.visualize()
```

31



```
animals_transported.compute().loc[('AU', 'Z')]
```

32

Importer reported quantity	1054.0
Exporter reported quantity	1013.9
Name:	(AU, Z), dtype: float64

Exercise: Looking at import and export discrepancies

In the CITES dataset there are a number of discrepancies, usually because for one reason or another records fail to be matched, or in some cases reported by either the exporter or the importer. You can consider there to be a discrepancy if the total number of imports for a country for each year does not match

the total number exported to that country each year. It can also be caused by differences in reported units from one country to another, and other data cleansing issues. Unfortunately this is very common. Calculate over time the average discrepancies in reporting. Is this getting better (plot the results to visualize the answer)?

```
# %load solutions/animal_transport.py
```

33

One other handy feature when reading multiple files is that you can include the path to the source file as a column. It will be loaded as a Category column. This is useful if the data source describes a property of the data that isn't present in the file. For example, looking at this rainfall data the filename describes the source measuring station:

```
rainfall = dd.read_csv(
    'Data/Rainfall/*.csv',
    include_path_column=True,
    header=None,
    names=['date', 'rainfall'],
    parse_dates=['date'],
    dayfirst=True,
).set_index('date')
rainfall
```

34

	rainfall	path
npartitions=4		
1930-09-01	float64	category[known]
00:00:00.000000000		
1946-11-08
02:46:22.229074944		
1965-03-26
20:14:32.247666272		
1984-07-25
02:23:59.999999872		
2007-08-31
00:00:00.000000000		

```
rainfall.pivot_table(columns='path', values='rainfall',
                     index='date', aggfunc='sum').compute().head()
```

35

path date	Data/Rain- fall/10010.csv	Data/Rain- fall/10011.csv	Data/Rain- fall/10012.csv	Data/Rain- fall/10013.csv
1930-09-01	0.00000	0.000000	0.000000	0.000000
1930-09-02	0.00000	0.000000	0.000000	0.000000
1930-09-03	0.00000	0.000000	0.000000	0.000000
1930-09-04	0.00000	0.570040	1.160273	0.923728
1930-09-05	3.06343	7.221712	1.768180	1.270434

Exercise: Measuring the response of bees to changes in temperature and humidity

Bees are extremely important for our environment. The HOBOS (HOneyBee Online Studies) is a honeybee project that came about in the city of Würzburg in 2006. Bees are equipped with microchips and along with other sensors a hive is monitored over time. In this case you have one year of data in the Data/bee-hive-metrics folder.

The datasets are as follows:

- **flow**: The number of bees leaving (negative numbers) or entering (positive numbers) the hive at any given time
- **humidity**: Level of humidity in the beehive (as %)
- **temperature**: The temperature from 13 sensors throughout the hive sampled over time, measured in °C
- **weight**: The total weight of the hive measured in kg

These datasets are all resampled at different periods. To start with read each dataset, parse the timestamp column and make it the index. Then resample each dataset to a 1 hour period (the aggregation function for the flow should be sum, the others should be mean).

Use the merge function on the Dask Dataframe and create a combined dataset. Finally, calculate the correlation before visualising the task graph and computing the result. What is the factor that the weight of the hive is *most* dependent on?

```
# %load solutions/dask_beans.py
```

36

24.4 dask.bag

The `dask.bag` is the dask implementation of operations on generic Python collections, such as `map`, `groupby`, `filter`, and `fold` (equivalent to the builtin `reduce` function). In general the bag will be slower than equivalent operations in `dask.array` and `dask.dataframe` but it is a much better choice when working with unstructured data.

```
import dask.bag as db
```

36

In general it's easy to create a bag from an existing Python collection:

```
db.from_sequence(['Henry', 'Hannah', 'Marcie', 'Rob', 'Ed'],
                 partition_size=2)
```

37

```
dask.bag<from_se..., npartitions=3>
```

But in general if you're creating a bag from a sequence you can already fit in memory in Python unless you need to significantly reduce the computation time by performing operations on each element in parallel processes, you are likely to be better off working with inbuilt Python functions. Happily it is easy to read directly from a file (or a `dask.delayed` function - see below):

```
alice = db.read_text('Data/alice_in_wonderland.txt.gz')
```

38

At this point the text will be read into Python as a collection of lines from the file (note you could read from multiple files in the same way as reading multiple files in `dask.dataframe` with a `glob` pattern).

As an example, you can use the standard map-reduce pattern to calculate the number of words in the file:

```
import re

word_matcher = re.compile('[^ \w]+')
```

39

```
def clean_words(line):
    line = word_matcher.sub(' ', line.upper())
    return ({
        'word': word,
        'count': 1
    } for word in line.split())

def binop(total, x):
    return total + x['count']

def combine(total1, total2):
    return total1 + total2

word_counts =
    ↵ alice.map(clean_words).flatten().foldby('word', binop,
    ↵ 0, combine, 0)
```

This deserves being broken down step by step:

1. The `map` on the bag applies the function `clean_words` to each line in the input bag. This function in turn cleans each line of punctuation characters, makes the text upper case (so you can ignore case for matching purposes), and returns the data in a common data structure for this kind of operation - a dictionary with the word and a count.
2. All `dask.bag.Bag` objects have a `flatten` method. This takes an iterable of iterables and chains it together in a single collection.
3. The `foldby` is the most complex. It combines a reduce and groupby operation by taking three functions as arguments:
 1. The first argument is a function that returns the key that will be used for grouping. Because it's such a common pattern to use the value from a dictionary(-like) object as this key, you can simply provide the key and it will be used in place of `lambda x: x['key']`
 2. The `binop`, or “binary operator” function takes a running total and produces a new total. The output type of the total should be the same as an input type. This is followed by a default value (in

- this case of zero)
3. The `combine` function is similar to the `binop` function, but instead of taking a total and a new element, takes two totals and combines them. This is also followed by a default value (again zero in this case)

```
word_counts.compute()
```

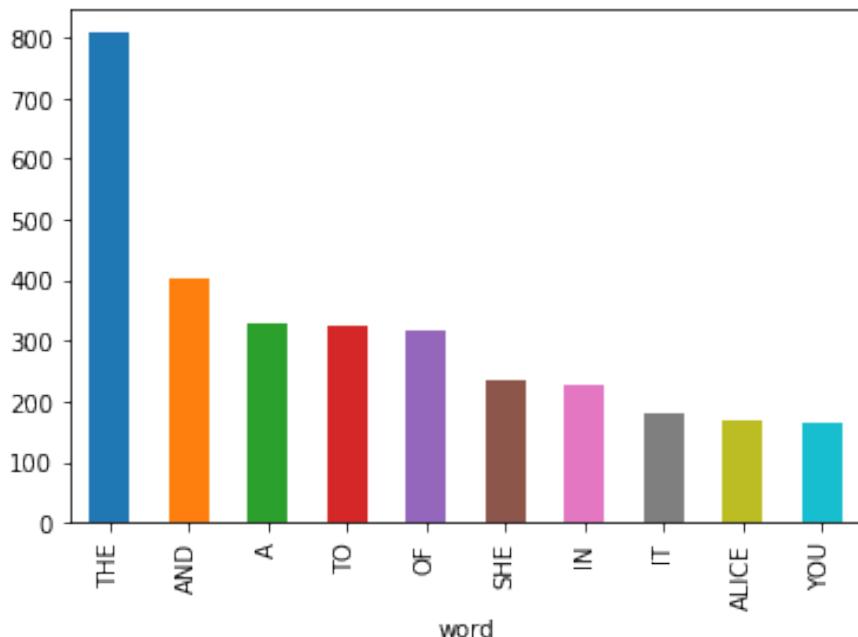
41

There are also convenience methods to (for example) convert a bag object to a `dask.dataframe` object.

```
word_counts.to_dataframe(columns=['word',
    'count']).set_index('word')['count'].nlargest(10).compute().plot(kind='bar')
```

181

```
<matplotlib.axes._subplots.AxesSubplot at 0x12faaf320>
```



Exercise: Counting duplicate files using map-reduce

You can calculate the md5 hash of file contents using the function below:

```
import hashlib

def md5(fname):
    """Read file chunk at a time to calculate MD5 has"""
    hash_md5 = hashlib.md5()
    with open(fname, "rb") as f:
        for chunk in iter(lambda: f.read(4096), b ""):
            hash_md5.update(chunk)
    return hash_md5.hexdigest()
```

42

```
md5('Data/alice_in_wonderland.txt.gz')
```

43

```
'06b9908b83aa9fb48e6dbe31ee76062f'
```

The same contents from different files will produce the same hash, which means you can use this to test that files have the same contents. Create a bag containing dictionaries in the form:

```
{'md5': md5(path), 'paths': [path, ]}
```

Where each path is a file from the Data folder (you can use `glob.glob` to search and recurse files). Write appropriate `bimap` and `combine` functions that instead of summing a number of files, append a list of files instead. Use `foldby` and `filter` to generate a collection of files which are duplicates.

```
# %load solutions/dask_file_duplicates.py
```

53

24.5 `dask.delayed`

The `dask.delayed` is a low level interface that allows you to decorate functions so that they execute “lazily” through the task graph. If you visualize the

dask.bag.from_sequence method above you might notice for example that it executes delayed functions under the hood.

```
from dask import delayed
```

59

The delayed decorator is most useful for situations where you have to call the same function over and over again and where results from the function are independent. For example, if you know Binet's formula you know to generate the n th number in a Fibonacci sequence you can follow the formula:

$$F_n = \frac{\phi^n - \psi^n}{\sqrt{5}}$$

where

$$\phi = \frac{1 + \sqrt{5}}{2}$$

and

$$\psi = -\frac{1}{\phi}$$

```
phi = (1 + 5**0.5) / 2
psi = -1 / phi

@delayed
def fibonacci(n):
    """Cast the result to an integer as because of floating
    → point error the solution is not exact"""
    return int(round((phi**n - psi**n) / (5**0.5)))
```

67

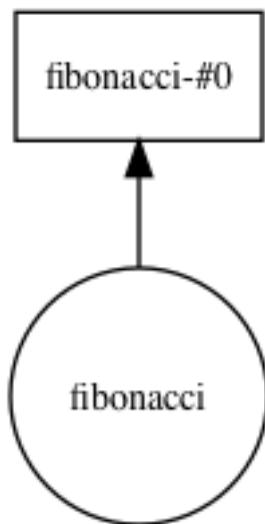
```
fibonacci(6)
```

70

```
Delayed('fibonacci-b200d725-167d-4a1a-a3be-0e8a821352fb')
```

```
fibonacci(6).visualize()
```

71



```
fibonacci(6).compute()
```

69

8

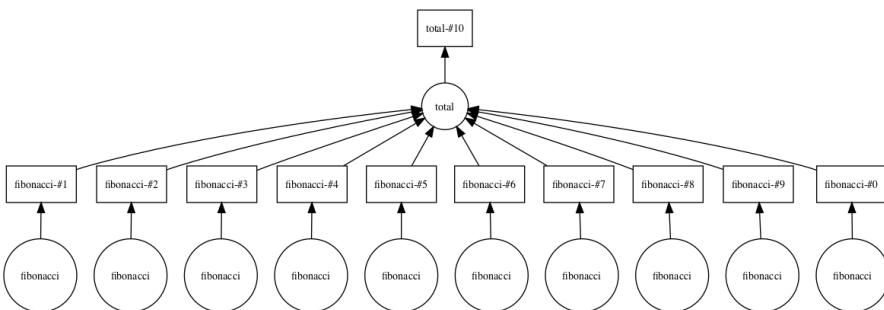
But passing the delayed function into other delayed functions you can form a more complex task graph:

```
@delayed
def total(*args):
    return sum(args)
```

72

```
total(*(fibonacci(i) for i in range(1, 11))).visualize()
```

73



Exercise: Writing a delayed task to call a web service

This API:

<https://places-api.pytoncharmers.com/location/nearest>

Returns the nearest place to a given pair of latitudes and longitudes given latitude and longitude parameters. For example:

```
import requests
```

74

```
url =
    'https://places-api.pytoncharmers.com/location/nearest'
params = {
    'latitude': -35.7,
    'longitude': 150.2
}
requests.get(url, params).json()
```

75

```
{'latitude': -35.6896187146,
'longitude': 150.207306681,
'name': 'Batemans Bay'}
```

Write two delayed functions: the first to generate random latitude and longitude pairs (note, latitude should be in the range -90 to 90 and longitude should be -180 to 180). The second should call the nearest location API and get the nearest location to each of these pairs. Generate 20 latitude and longitude pairs and pass them to the nearest location

function. You can call these functions as a single computation using `dask.compute(*delayed_fn_collection)`.

```
# %load solutions/dask_nearest_places.py
```

99

24.6 Schedulers

Dask supports several schedulers:

- synchronous (for debugging)
- threaded
- multiprocessing
- distributed (EC2 et cetera: separate projects)

```
%timeit da.arange(10**6, chunks=(10**6,))[:2].compute()
```

9

The slowest run took 6.57 times longer than the fastest. This ↴ could mean that an intermediate result is being cached.
100 loops, best of 3: 1.99 ms per loop

You can also specify the scheduler with these keywords rather than having to use the “get” method:

- "threads" - the default for `dask.array` and `dask.dataframe`
- "synchronous" - process the data one at a time
- "processes" - the default for `dask.bag` and `dask.delayed`

The two most common ways to specify the scheduler to dask are:

```
from dask.context import set_options  
  
with set_options(scheduler='threads'):  
    pi.compute()
```

57

```
pi.compute(scheduler='threads')
```

58

3.1426784

```
%timeit pi.compute(scheduler='threads')
```

55

208 ms ± 8.44 ms per loop (mean ± std. dev. of 7 runs, 1 loop
↳ each)

Notice that if possible the threaded processor should be used for local computation. It works especially well where functions do not block with the GIL. In cases where the GIL is an issue you should use the process scheduler instead but be aware - especially on Windows - there is a cost to starting a new process and duplicating the current scope.

```
# Notice that this takes ages:  
# %timeit pi.compute(get=dask.multiprocessing.get)  
# Why?!
```

1

24.7 Distributed processing with Dask

First you must have the `distributed` package installed. It can be installed if not already available with either `pip` or `conda`:

```
pip install -U distributed  
conda install --yes distributed
```

Launching a scheduler and one or more workers

Now open the terminal / command prompt and type `dask-scheduler`.

Make a note of the IP address that the scheduler is at (e.g. `127.0.0.1:8786`).

Then start another terminal / command prompt and type `dask-worker` `$IP_ADDRESS` (e.g. `dask-worker 127.0.0.1:8786` with the example above).

Now we are ready to make use of all the workers attached to the given scheduler. For example:

```
from distributed import Client
client = Client('localhost:8786') # or whatever IP address
    ↵ and port the scheduler node is accessible on
```

101

You can then call `compute` on the client and pass in a delayed task graph for it to work on the scheduler:

```
future = client.compute(total(*(fibonacci(i) for i in
    ↵ range(1, 11))))
```

107

The future object you return has a `status` parameter to check the current execution status of execution:

```
future.status
```

108

```
'pending'
```

Of course, the Jupyter representation will show you this status as well:

```
future
```

109

Future: total status: pending, key: total-fff5cd9e-ee27-429d-82f0-a9c90e2fc9c2

To get the result back you can call the blocking operation `result()` on the future object:

```
future.result()
```

110

143

Note that the client processes are independent processes from the process where you are creating the task graph. This means if your task is reading data

from some parameter or file from the disk this must be available generally, not just to your local process. This is especially important when your clients are across multiple machines connecting to the dask-scheduler:

```
future = client.compute(word_counts) # paths used in word  
↪ counts is from a locally defined function  
future
```

116

Future: finalize status: error, key: finalize-bf8ef39116979dac10161b1359970ac4

```
future.exception()
```

117

```
KeyError('paths')
```

24.8 Under the hood: task graphs

Dask task graphs are stored internally as dictionaries. You can read more about their structure here: <http://dask.pydata.org/en/latest/graphs.html>

Here is an example of the graph via the `.dask` attribute of a Dask object:

```
import dask.array as da
```

118

```
x = da.ones((5, 15), chunks=(5, 5))  
type(x)
```

119

```
dask.array.core.Array
```

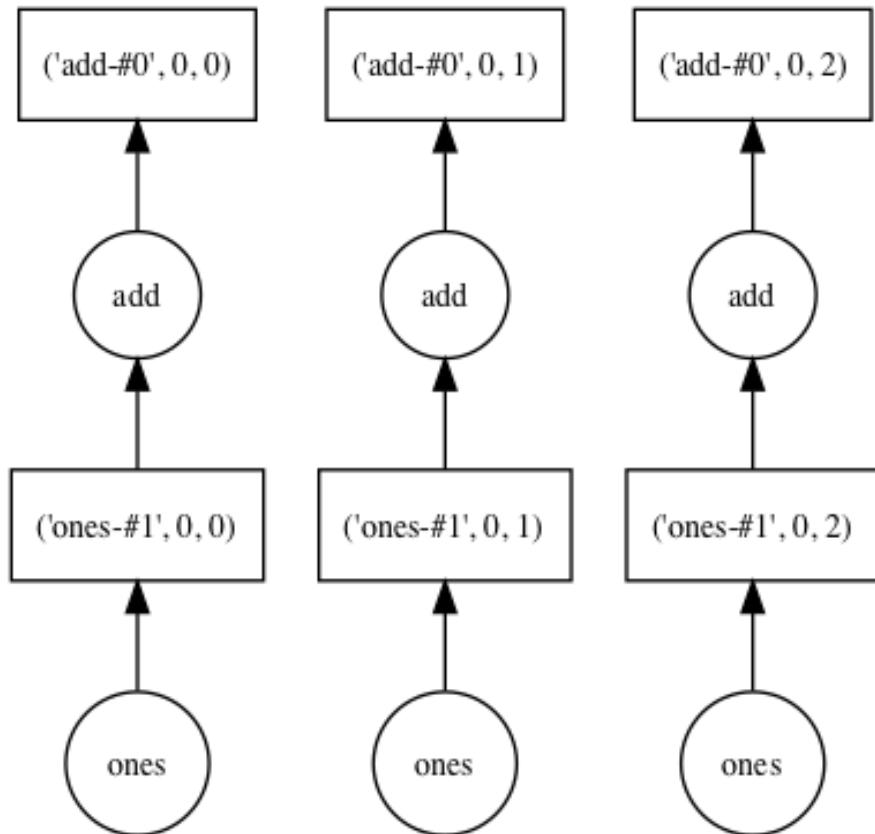
```
x.dask
```

120

```
<dask.sharedict.ShareDict at 0x120b49b38>
```

```
d = (x + 1).dask  
from dask.dot import dot_graph  
dot_graph(d)
```

121



Chapter 25

Web APIs

An Application Programming Interface (API) is a contract between two systems: if the client sends a request in a specific format it will always receive a response from the service in a specific format or trigger a defined action. APIs are not limited to the internet but in many cases when people refer to an API they are referring to RESTful web APIs with communication between a client and a server over the web. This chapter shows you how to interact with web APIs in Python using the `Requests` library.

25.1 HTTP in a nutshell

The protocol of the web is HTTP. This is used for serving up traditional web pages and also web services exposed via web APIs.

HTTP is a simple client-server protocol:

Your web browser is an HTTP client. So is the Python `requests` library.

HTTP is a simple language (protocol). Its most important verb is GET. Most pages display in your browser from GET requests. We can issue a GET request from Python using `requests` as follows:

```
import requests
```

2

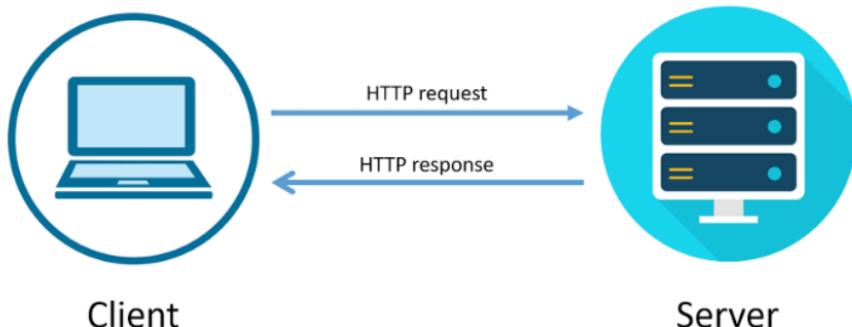


Figure 25.1: HTTP client-server protocol

```
url = "http://www.google.com"
response = requests.get(url)
print(response.text[:200])
```

```
<!doctype html><html itemscope=""
  itemtype="http://schema.org/WebPage"
  lang="en-AU"><head><meta content="text/html;
  charset=UTF-8" http-equiv="Content-Type"><meta
  content="/images/branding/googleg/1x/
```

What we see above is the first 200 characters returned by the HTTP server at `google.com`.

Making a GET request from `www.google.com` only returns the web content necessary for rendering the google home page in our browser. Typically, however, we are using APIs in order to acquire data. To see an example of this, we shall use the public World Bank Open Data API (documentation available at <https://datahelpdesk.worldbank.org/knowledgebase/topics/125589-developer-information>).

We're going to use this API to retrieve a list of all the income levels with which the World Bank classifies countries. The URL we need to use is `https://api.worldbank.org/v2/incomeLevels`.

Note the v2 after `api.worldbank.org`: this signifies that we're access-

ing version two of the API and also lets us know that this (as well as any decent API) is concerned with stability such that we can be confident that the behaviour of this API, so long as we use v2, will not change or break our code.

Retrieving our desired data works much the same as above.

```
url = 'https://api.worldbank.org/v2/incomeLevels'  
response = requests.get(url)  
print(response.text)
```

2

```
ï»¿<?xml version="1.0" encoding="utf-8"?>  
<wb:IncomeLevels page="1" pages="1" per_page="50" total="7"  
  ↪ xmlns:wb="http://www.worldbank.org">  
  <wb:incomeLevel id="HIC" iso2code="XD">High  
    ↪ income</wb:incomeLevel>  
  <wb:incomeLevel id="INX" iso2code="XY">Not  
    ↪ classified</wb:incomeLevel>  
  <wb:incomeLevel id="LIC" iso2code="XM">Low  
    ↪ income</wb:incomeLevel>  
  <wb:incomeLevel id="LMC" iso2code="XN">Lower middle  
    ↪ income</wb:incomeLevel>  
  <wb:incomeLevel id="LMY" iso2code="XO">Low & middle  
    ↪ income</wb:incomeLevel>  
  <wb:incomeLevel id="MIC" iso2code="XP">Middle  
    ↪ income</wb:incomeLevel>  
  <wb:incomeLevel id="UMC" iso2code="XT">Upper middle  
    ↪ income</wb:incomeLevel>  
</wb:IncomeLevels>
```

From the first line you can see that the data we have gotten back is in XML format which has become unpopular in recent years and often poses data parsing problems that are unnecessarily complex.

The more popular and user-friendly format we want to use is JSON, which turns out to be expressed in a way very similar to the way python creates lists and dictionaries!

To get data back in JSON format we will have to add `format=json` to our

URL as follows.

```
url = 'https://api.worldbank.org/v2/incomeLevels?format=json'  
response = requests.get(url)  
print(response.text)
```

```
[{"page": "1", "pages": "1", "per_page": "50", "total": "7"},  
 ↴ [{"id": "HIC", "iso2code": "XD", "value": "High  
 ↵ income"}, {"id": "INX", "iso2code": "XY", "value": "Not  
 ↵ classified"}, {"id": "LIC", "iso2code": "XM", "value": "Low  
 ↵ income"}, {"id": "LMC", "iso2code": "XN", "value": "Lower middle  
 ↵ income"}, {"id": "LMY", "iso2code": "XO", "value": "Low & middle  
 ↵ income"}, {"id": "MIC", "iso2code": "XP", "value": "Middle  
 ↵ income"}, {"id": "UMC", "iso2code": "XT", "value": "Upper middle  
 ↵ income"}]]
```

Immediate you should be able to see the resemblance between this format and our python lists and dictionaries. Hopefully by comparing with the above XML output you can confirm that the same raw data is contained in both outputs.

But, so far our data is just a string, not an actual list or dictionary that we can easily analyse with python. To parse the JSON the raw data into a python object, instead of using `response.text`, we must use the built-in function `json()`, which will return a python list or dictionary as needed.

```
income_levels = response.json()  
print(income_levels[1][0])
```

```
{'id': 'HIC', 'iso2code': 'XD', 'value': 'High income'}
```

```
income_level_ids = []  
  
for il in income_levels[1]:  
    income_level_ids.append(il['id'])  
print(income_level_ids)
```

```
['HIC', 'INX', 'LIC', 'LMC', 'LMY', 'MIC', 'UMC']
```

Exercise

The World Bank collects data from many sources. The API allows us to retrieve a complete list of all the sources used. To access this the URL used above must have `incomeLevels` replaced by `sources`. Adapt the above code to obtain a list of the World Bank's sources. Don't forget to retrieve the data in JSON format.

25.2 URLs and Parameters

You will notice that the income levels URL contained a question mark and that it was after this question mark that we specified the response format as JSON. This is a common pattern with APIs.

Everything before the question mark is a URL or address or *endpoint* and serves to define a specific API or target from which one can make requests. Everything after the question mark provides additional *parameters* or arguments to the API to further specify how the request is going to work. In the above example we used the `format` parameter to specify the format of our response data.

The parameters that an API accepts and processes is arbitrary and must be observed from the API's documentation. For instance, a number of additional parameters, many of which relate to specifying time periods, that are available for the World Bank API are documented at <https://datahelpdesk.worldbank.org/knowledgebase/articles/898581-api-basic-call-structures>.

The recommended way of handling parameters with the `requests` library is to express them as a dictionary and pass this dictionary to the request as the `params` argument. This avoids the error-prone and long URLs that parameters can cause.

```
url = 'https://api.worldbank.org/v2/incomeLevels'  
params = {'format': 'json'}  
  
response = requests.get(url, params=params)  
print(response.json())
```

6

```
[{'page': '1', 'pages': '1', 'per_page': '50', 'total': '7'},
 ↵  [{id: 'HIC', 'iso2code': 'XD', 'value': 'High income'},
 ↵  {id: 'INX', 'iso2code': 'XY', 'value': 'Not
 ↵  classified'}, {id: 'LIC', 'iso2code': 'XM', 'value':
 ↵  'Low income'}, {'id': 'LMC', 'iso2code': 'XN', 'value':
 ↵  'Lower middle income'}, {'id': 'LMY', 'iso2code': 'XO',
 ↵  'value': 'Low & middle income'}, {'id': 'MIC', 'iso2code':
 ↵  'XP', 'value': 'Middle income'}, {'id': 'UMC', 'iso2code':
 ↵  'XT', 'value': 'Upper middle income'}]]
```

Countries and Indicators

The two main endpoints provided by the World Bank API for retrieving data are `/countries` (i.e. <https://api.worldbank.org/v2/countries>) and `/indicators` (which specify the metric of interest). Generally these two are combined to obtain the data one desires.

The remaining endpoints are primarily for retrieving secondary, descriptive or metadata. They are `/incomeLevels`, `/lendingTypes`, `/sources` and `/topics`.

Listing Sources, Indicators and Countries

The World Bank's indicators are obtained from a limited set of sources typically with a general focus. Searching for an indicator of interest can therefore be done by retrieving the available sources and then the indicators under a particular source.

```
# Getting list of sources

resp = requests.get('https://api.worldbank.org/v2/sources',
                     params={'format':'json',
                             'per_page':'100'})
```

```
resp.json()[1][:2]
```

```
[{'id': '1',
 'lastupdated': '2019-10-23',
 'name': 'Doing Business',
 'code': 'DBS',
 'description': '',
 'url': '',
 'dataavailability': 'Y',
 'metadataavailability': 'Y',
 'concepts': '3'},
 {'id': '2',
 'lastupdated': '2020-07-01',
 'name': 'World Development Indicators',
 'code': 'WDI',
 'description': '',
 'url': '',
 'dataavailability': 'Y',
 'metadataavailability': 'Y',
 'concepts': '3'}]
```

Sometimes a function like that below can be useful for quickly isolating the relevant data from the long and rich lists APIs return

```
def get_fields(data, fields):
    '''Retrieve specified fields for each entry in data'''
    data_list = []
    for entry in data:
        entry_list = []
        for f in fields:
            entry_list.append(entry[f])
        data_list.append(entry_list)

    return data_list
```

9

```
sources_clean = get_fields(resp.json()[1], ['id', 'code',
                                             'name'])

print(sources_clean[:3])
```

10

```
[['1', 'DBS', 'Doing Business'], ['2', 'WDI', 'World
    ↵ Development Indicators'], ['3', 'WGI', 'Worldwide
    ↵ Governance Indicators']]
```

Exercise

Using the indicators API (URL: <https://api.worldbank.org/v2/indicators>) and the parameter `source` along with the `id` of the source we're interested in, we can obtain a list of all the indicators originating from the specified source.

Retrieve a list of available sources. Select a source of interest, retrieve the indicators available from it and select one of those of interest. Don't forget to use the parameters `per_page` and/or `page` to obtain all possible results if necessary.

Extension

Using the `/countries` endpoint (i.e. <https://api.worldbank.org/v2/countries>) write a function that determines what the `id/isocode` or the `iso2code` for a particular country (such as 'Aruba').

Using either string methods or regular expression, can you write a function that allows you to easily search the names of the various entries the API gives back to you for a particular word like *population*?

Getting Country Data

To retrieve data for a specific indicator either for all countries or a specific country requires combining the `/indicators` and `/countries` endpoints, along with the specific country and indicator we're interested in, into a single URL.

For example, to retrieve data for the population indicator `SP.POP.TOTL`, found under the source `World Development Indicators`, for the United States or USA, would require the following URL:
<https://api.worldbank.org/v2/countries/USA/indicators/SP.POP.TOTL>

This pattern of using URLs instead of parameters to query an API is not uncommon. The World Bank have combined both parameters and complex URLs

in the design of their API. Fortunately python's string functionality is very useful for such tasks, especially `f-strings`.

```
url = 'https://api.worldbank.org/v2'  
country = 'USA'  
indicator = 'SP.POP.TOTL'  
url = f'{url}/countries/{country}/indicators/{indicator}'  
  
resp = requests.get(url, params={'format':'json'})  
  
pop_data = resp.json()  
  
print(pop_data[1][:3])
```

```
[{'indicator': {'id': 'SP.POP.TOTL', 'value': 'Population, total'}, 'country': {'id': 'US', 'value': 'United States'}, 'countryiso3code': 'USA', 'date': '2019', 'value': 328239523, 'unit': '', 'obs_status': '', 'decimal': 0}, {'indicator': {'id': 'SP.POP.TOTL', 'value': 'Population, total'}, 'country': {'id': 'US', 'value': 'United States'}, 'countryiso3code': 'USA', 'date': '2018', 'value': 326687501, 'unit': '', 'obs_status': '', 'decimal': 0}, {'indicator': {'id': 'SP.POP.TOTL', 'value': 'Population, total'}, 'country': {'id': 'US', 'value': 'United States'}, 'countryiso3code': 'USA', 'date': '2017', 'value': 324985539, 'unit': '', 'obs_status': '', 'decimal': 0}]
```

Exercise

Lets write a function that returns the GDP (or your preferred indicator) for any given country.

A good GDP indicator is the “*GDP (constant 2010 US\$)*” (id: `NY.GDP.MKTP.KD`).

Adapt the code above into a function that should operate like the following:

```
USA_gdp = get_gdp('USA')
```

```
print(USA_gdp) # 18318722096911
```

Don't forget that countries need to be specified in either 3 or 2 letter iso format (e.g. BR or BRA for Brazil).

Extensions

Notice that the data returned lists historical yearly data for GDP. Read the API documentation to determine how to use parameters to specify a date (*hint: date=2000*) or to limit the number of values returned (*hint: MRV=1*).

The API is capable of accepting multiple countries in the URL so long as they are separated by a semi-colon (;). Modify your function to be able to take a list of countries as an argument. You will need to format the multiple countries appropriately within the function for the URL. You will also need to alter the output of your function to accommodate the multiple values you will have.

The API can also accept multiple indicators so long as they too are separated by semi-colons like the countries. When doing so, however, you will often need to also provide the source of the indicators as a parameter (the API can get confused otherwise). Presuming all indicators come from source id: 2, can you make a function that accepts and manages `indicators`, `countries` and `dates` arguments?

Data Analysis with Pandas

Often the best way to get a handle on large and rich data structures like those provided by APIs is to find a way to get your data into a pandas dataframe and use all of the tools provided by that library.

Let's try to find the correlation between two indicators.

To begin, we must first retrieve data efficiently. From the exercises above, let us settle on a utility function for retrieving data like so:

```
def get_indicator_data(indicators, date='2019',
    ↵ countries='all', source='2', per_page=1000):
    '''Retrieve data for specified countries and
    ↵ indicators'''
```

12

```
url = 'https://api.worldbank.org/v2'

if isinstance(countries, list):
    countries = ';' .join(countries)

if isinstance(indicators, list):
    indicators = ';' .join(indicators)

url = f'{url}/countries/{countries}/indicators/」
    ↵ {indicators}'

parameters = {
    'format': 'json',
    'date': date,
    'source': source,
    'per_page': per_page
}

resp = requests.get(url, params=parameters)
resp_data = resp.json()

return resp_data
```

To give it a spin, let's get data for the “*GDP per capita (constant 2010 US\$)*” indicator (id: NY.GDP.PCAP.KD) and the “*School enrollment, primary (% gross)*” (id: SE.PRM.ENRR), for the United States and for the year 2012 as not all data is reliably updated.

```
data = get_indicator_data(['SE.PRM.ENRR', 'NY.GDP.PCAP.KD'],
    ↵ countries='usa', date='2012')
print(data)
```

13

```
[{'page': 1, 'pages': 1, 'per_page': 1000, 'total': 2,
 ↵ 'sourceid': None, 'lastupdated': '2020-07-01'},
 ↵ [{"indicator": {"id": "SE.PRM.ENRR", "value": "School
 enrollment, primary (% gross)"}, "country": {"id": "US",
 "value": "United States"}, "countryiso3code": "USA",
 "date": "2012", "value": 99.16532, "unit": '',
 "obs_status": '', "decimal": 0}, {"indicator": {"id": "NY.GDP.PCAP.KD",
 "value": "GDP per capita (constant 2010
 US$)"}, "country": {"id": "US", "value": "United States"}, "countryiso3code": "USA",
 "date": "2012", "value": 49603.2534736283, "unit": '',
 "obs_status": '', "decimal": 1}]]
```

Now, to create a flat table of data from this, we're going to have to flatten the nested dictionaries under `indicator` and `country`. A function like that below which loops through each entry and creates a new simpler dictionary will do the trick.

```
def clean_wbdata(data):
    new_data = []
    for value in data:
        new_value = {
            'indicator': value['indicator']['value'],
            'country': value['country']['value'],
            'date': value['date'],
            'value': value['value']
        }
        new_data.append(new_value)

    return new_data
```

```
clean_data = clean_wbdata(data[1])
print(clean_data)
```

```
[{'indicator': 'School enrollment, primary (% gross)',  
 ↵ 'country': 'United States', 'date': '2012', 'value':  
 ↵ 99.16532}, {'indicator': 'GDP per capita (constant 2010  
 ↵ US$)', 'country': 'United States', 'date': '2012',  
 ↵ 'value': 49603.2534736283}]
```

Now that the structure of the data is flat and square, pandas is capable of creating a DataFrame from it.

```
import pandas as pd
```

16

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

17

	indicator	country	date	value
0	School enrollment, primary (% gross)	United States	2012	99.165320
1	GDP per capita (constant 2010 US\$)	United States	2012	49603.253474

Given the structure of this data and our purpose, transforming it from long-form to wide-form will be helpful. The df.pivot function performs this task.

```
df = df.pivot(index='country', columns='indicator',  
 ↵ values='value')  
df
```

18

country	GDP per capita (constant 2010 US\$)	School enrollment, primary (% gross)
United States	49603.253474	99.16532

Exercise

Retrieve data as done above but for all countries. You should end up with a much larger dataframe than the one above. Find the correlation between the

two indicators.

Perform the same analysis but with “*Mortality rate, infant (per 1,000 live births)*” (id: SP.DYN.IMRT.IN) indicator rather than previous schooling indicator (id: SE.PRM.ENRR). What’s the correlation between infant mortality and GDP per capita?

Extensions

Can you wrap your workflow up into a function that returns a dataframe from just the indicator, country and data arguments like the function above? You could use this to find the correlation between other indicators of interest

Each country is classified as being of a particular income level. This data is available under the /countries endpoint and could be joined to our indicator dataframes using the `join` or `merge` method. From this, the distributions of the indicators could be compared between the income levels, especially through appropriate plots.

25.3 POST

Above we saw GET requests. Although this is perhaps the most common HTTP verb in web APIs, another important one is POST.

POST corresponds to submission of data, traditionally via a HTML form. A POST request is expected to change the state on the server (whereas GET does not).

GET vs POST in detail

Parameters

POST

GET

Data Parameters

It does not store and includes the data parameters in the URL or anywhere on the client system

It includes the data parameters in the URL which can store in the browser

history and the security of the client can be compromised.

Bookmark

As the data does not include in URL so there is no concept of Bookmark

The get request data can be Bookmarked

Reload the page

The data will be resubmitted.

The page will be re-executed but not re-submitted, it does not send any request to the host because it is saved in the cache memory.

Security

POST provide security over the data

It does not provide any security

Cache

No cache

There can be caches

Data length

There is no limit for data length

It has a limit of 2048 characters.

Data type

All data types are allowed

Only accept ASCII characters

Data visibility

Data is not visible to everyone

Data is visible to everyone

Use

We use the POST method with sensitive data like password

GET used with insensitive data.

25.4 Using the Telstra messaging APIs

This is an extended example: using the Telstra Messaging APIs to send yourself an SMS or MMS.

It illustrates:

- making sense of API docs (and/or OpenAPI spec)
- authenticating with oAuth
- POST requests

To begin, go to <http://dev.telstra.com> and log in. Look up your “client key” and “client secret”.

```
# Example:  
key = '3SiryjmCLrQ08EXJyl0QHSJCqSqUW3AP'  
  
from getpass import getpass  
# Paste in your client secret as keyboard input here:  
secret = getpass('Client secret:')
```

1

Exercise steps

1. Look at the Messaging API docs at dev.telstra.com
2. Try sending yourself a text message via SMS!
3. Challenge exercise: try sending yourself an image via MMS. Tip: base64 encode your PNG data (get help on the `base64.b64encode()` function)

Step 1: authenticate

```
import requests  
from getpass import getpass  
  
client_key = '3SiryjmCLrQ08EXJyl0QHSJCqSqUW3AP'  
client_secret = getpass('Client secret:')
```

1

```
auth_url = 'https://tapi.telstra.com/v2/oauth/token'
auth_params = {'grant_type': 'client_credentials',
               'client_id': client_key, 'client_secret':
               ↴ client_secret, 'scop': 'NSMS'}
auth_response = requests.post(auth_url, data=auth_params)
print(auth_response.json())
```

1

```
# Question: how to check the expiry time
# import datetime as dt
# start = dt.datetime.now()
# start + dt.timedelta(seconds=int(auth_response.json([
#   ↴ )['expires_in'])))
```

1

```
token = auth_response.json()['access_token']
```

1

Step 2: provision a phone number

```
provision_url = 'https://tapi.telstra.com/v2/messages/'
                ↴ provisioning/subscriptions'
headers = {'Authorization': f'Bearer {token}'}

provision_response = requests.post(provision_url,
                                    ↴ headers=headers, json={})

address = provision_response.json()['destinationAddress']
```

Step 3: send yourself an SMS

Read the docs on sending an SMS. The CURL examples may be the easiest to follow.

For example:

```
params = {'to': '+61412345678',
          'body': 'Test SMS message!',
          'from': address}

url = 'https://tapi.telstra.com/v2/messages/sms'

response = requests.post(url, json=params, headers=headers, pro-
                           tected=True)

print(response.text)
```

Challenge exercise: send yourself an MMS with a picture

You could try sending yourself an image of your own, or one from the data folder, or one you download. An example: an XKCD comic.

- Tip 1: Fetch an image using `requests.get(image_url).content` or read it in from disk (in bytes mode).
- Tip 2: try base64-encoding the bytes of your image

Here is how base64 encoding works in Python:

```
import base64
base64.b64encode(image_data_as_bytes).decode()
```

1

Chapter 26

Creating web apps with Flask

26.1 Getting started

Flask is a lightweight web framework that allows you to quickly and easily create dynamic websites and APIs.

Flask applications can grow to be quite large, but today we are going to create a small Flask app that consists of just a few files. There is a full set of documentation on the USB stick, which includes a fantastic tutorial on using Flask and most of the features that are used.

Today, we will be creating an API that performs several functions for our code. Follow these steps to get started:

1. Create a folder in the home directory called “myapp”.
2. Inside that folder, create a text file called `myapp.py` (Right click and choose New... Empty File, make sure to remember the extension!).
3. Open the text file, and add the following code:

```
from flask import Flask  
  
app = Flask(__name__)
```

1

```
@app.route("/")
def index():
    return "Hello world!"

if __name__ == '__main__':
    app.run(debug=True, port=5000)
```

In line 1, we import Flask, and in line 3, we create a new Flask application. The `index` function returns a simple string, and by using the decorator (marked up using the @ symbol), we tell flask to link this to the “/” route – i.e. the root.

Finally, using the `if __name__ == '__main__':` line, we tell python that “when this file is run as a script”, do the following code. At this point, it calls the `run()` function on the app.

To test it out, use the terminal and change to the `myapp` directory:

```
cd myapp
```

Then, run the python script:

```
python myapp.py
```

After it starts up, it will give you the URL to go to, to see your web server. Navigate to this location (probably `http://127.0.0.1:5000/`) in your web browser, to see the “Hello World!” message.

Exercise Set 1

1. Create a new function called `hello`, that is linked to the `/hello` endpoint (rather than `/`). Change the message to include your name.
2. Test it out by restarting the server (use `CTRL + C` to stop the current server, then run the command again) and going to `http://127.0.0.1:5000/hello`
3. Change the line `app.run()` to `app.run(debug=True)`, then restart the server. This will tell Flask to look for any changes to the app, and

automatically restart the server. You now won't need to restart the server every time you make a change.

4. Create a new function at the endpoint /hello/<username>, and use the following code for the function: `python def hello_world(username): return 'Hello {}!'.format(username)` Navigate to `http://127.0.0.1:5000/hello/Your-Name` to see the result. Parameters are picked up by name from the URL using the <> brackets, and translated to python variables. These can be used as any other python variable.
5. Create a new function with endpoint /convert/<GiB> that uses the GiB to bytes converter we wrote on Day 1 (if you don't have this code, see the file `solutions/gibibytes.py`). **Tip:** Make sure to convert the GiB value to a float first!

```
%load solutions/flask_set1/myapp.py
```

1

Exercise Set 2

1. Using the `requests` library, call your Flask app's convert function from within a Jupyter notebook and print out the result.
2. What happens if you pass invalid parameters (i.e. a string instead of a number for GiB)?
3. You can test if a parameter is a valid number by using this function: `python def is_number(s): try: # Try convert to a number -- if this fails, a ValueError occurs float(s) # Above code succeeded -- s must be a number, or able to be interpreted as one return True except ValueError: # float conversion failed, so not a number return False`

Using this function, update your `myapp` code to test if the parameter is a valid number, and return a 400 error if it is not:

```
from flask import Flask, abort
```

```
abort(400) # Immediately throws a 400 error
```

26.2 Templates

Flask also has a templating library that can be used to generate HTML responses. To test this out, we will create a login page.

1. Create a folder called `templates` in your `myapp` folder
2. Within that folder, create a file called `login.html` with the following code (it isn't a complete HTML file, but it will render properly):

```
<form action="/login" method=POST>
    Username: <input type=text name=username><br>
    Password: <input type=password name=password><br>
    <input type=submit>
</form>
```

1

```
# Add the following endpoint to your myapp.py file:
from flask import render_template, request

@app.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        if request.form['username'] == "user" and
           request.form['password'] == "swordfish":
            # Valid username and password
            return "good"
    else:
        # Invalid username/password combination
        return "bad"
else:
    # Show the user the login form
    return render_template('login.html')
```

20

```
%load solutions/flask_set3/myapp.py
```

1

How this code works:

1. If the page `/login` is viewed in the web browser, the `login.html` template is rendered and shown. Try it at `http://127.0.0.1:5000/login`
2. If you submit this form, a POST request happens to `/login`, and the `request.method == "POST"` condition is True.
 - If the username and password is correct, the word “good” is displayed.
 - Otherwise, “bad” is displayed

Exercise Set 3

1. Create a `hello.html` template, that displays “Hello user!” when viewed.
2. Update the login code to render this template when the user successfully logs in.
3. Replace the endpoint at `/hello` to render this same template.

```
%load solutions/flask_set3/myapp.py
```

1

```
%load solutions/flask_set3/templates/hello.html
```

1

26.3 Sessions

Recoding session variables and logging in

We can create a session in Flask, which will keep track of whether the user has logged in. To do this, we first import the `session` object. Place the following code at the top of your `myapp.py` file:

```
from flask import session
```

1

Add the following line of code to the login function, *just above* the line that renders the hello.html template on a successful login:

```
session['username'] = request.form['username']
```

1

For sessions to work, you'll need to create a secret key. Just above the `app.run` line, add this line (indented one level):

```
app.secret_key = 'A0Zr98j3yX R~XHH!jmN]LWX/,?RT'
```

1

In production, you should set a different key to this one (you could even set your own here by choosing random letters/numbers/symbols).

This adds a new variable, called `username` to the session object. If this doesn't exist, then we say that no user is logged in. We can use this fact to easily create a logout endpoint by just removing this variable (pop will return and remove the variable with the given key in the session dictionary object):

```
from flask import redirect

@app.route("/logout")
def logout():
    session.pop("username")
    return redirect("/")
```

1

This will log out the user (if they exist), and then redirect them to the root page.

Within a function, you can check if there is a user logged in by using this code:

```
if 'username' in session:
    # User is logged in, perform some action for members
else:
    # No user is logged in, probably redirect to the login
    # page
    return redirect("/login")
```

1

Exercise Set 4

1. Update the /hello endpoint to only show when the user is logged in.
2. Alter the hello.html template to the following code: `html Hello {{ username }}!` This allows you to use a variable in the template.
3. To pass a variable, update the render_template and pass the parameters you want to set. In this case, it is the username parameter: `python return render_template('hello.html', username=session['username'])`

```
%load solutions/flask_set4/myapp.py
```

1

```
%load solutions/flask_set4/templates/hello.html
```

1

Flash messages

Flash messages are a quick way to send information from the server to the user. For example, when their username or password is incorrect.

Add the following import to your myapp.py file:

```
from flask import flash
```

1

You can then call `flash` with a message. This will add it to a stack of messages that can be displayed when a template is rendered. Replace the code that occurs when an invalid login is attempted with the following:

```
flash("Incorrect login, try again")
return redirect("/login")
```

1

At this point, nothing happens on the user's end. We can, however, show flash messages by adding it to our template. Replace the (entire) contents of login.html with the following:

```
{% with messages = get_flashed_messages() %}  
    {% if messages %}  
        <ul class=flashes>  
            {% for message in messages %}  
                <li>{{ message }}</li>  
            {% endfor %}  
        </ul>  
    {% endif %}  
{% endwith %}  
  
<form action="/login" method=POST>  
Username: <input type=text name=username><br>  
Password: <input type=password name=password><br>  
<input type=submit>  
</form>
```

What this does is looks for the presence of flash messages, and then displays them in a list (you don't need to display them like this, you can do whatever rendering you want). The login page then displays the form as before.

Exercise Set 5

1. Using the requests library, log into your application, and then get the contents of the /hello endpoint. Print out the text of the response. To do this, you'll need to use a Session object in the requests library. See the first example on this page: <http://docs.python-requests.org/en/v2.0-0/user/advanced/>

Hint: instead of using .get, you'll need to use .post and send through the credentials as data.

2. Change the /hello endpoint to return a JSON response. You can do this by creating a dictionary of values and then using Flask's jsonify():

```
data = {}  
data['logged_in'] = True
```

```
data['username'] = session['username']
data['message'] = "Hello
    ↵   {}".format(session['username'])
return jsonify(**data)
```

You will need to import `jsonify` from the `flask` package.

3. Update your code to call your endpoint (from question 1 in this set) to print the JSON object instead of the text.
4. Update the `/hello` endpoint to accept a GET parameter that is a number. If given, this repeats the greeting that many times. You'll need to convert the parameter to an integer first (using `int()`), and you can get the parameter using: `request.args.get("number", 1)`

This will also set the default value to 1, if the parameter wasn't given.

Test it out by going to `http://127.0.0.1:5000/hello?number=6` in your web browser.

```
%load solutions/flask_set5/myapp.py
```

1

```
%load solutions/flask_set5/templates/login.html
```

1

Extended Exercise

Create an endpoint based on the log parsing exercises from day 1. The endpoint should do the following: 1. List the different interfaces 2. List the ones that are down 3. Take a parameter that searches through the log for instances (i.e. grep)

Then create documentation for your API using Swagger.

Chapter 27

Best practices in Flask

27.1 Handling URL parameter types

When you specify a URL route in Flask you have a few options for how to handle the *type* of data handled by that path.

Let say that we had an endpoint that must only accept an integer in the path, for example:

```
/invoice_number/<num>/
```

If you made an endpoint that looked like this:

```
invoices = {  
    1: "invoice 1",  
    2: "invoice 2",  
}  
  
@app.route("/invoice_number/<num>/")  
def get_invoice(num):  
    """Returns the details for the given invoice number"""  
    invoice_info = invoices[num]  
    return f"Invoice number {num}: {invoice_info}"
```

1

There's a few subtle bugs that are present here. If we didn't do some processing first we would have a `KeyError` any time that someone didn't provide a number.

URL Variable rules

A good way to handle this is to use Flask's path parameter type support to automatically 404 on any URL path items that are the wrong type. This gives us the benefits of not having to write the type checking code ourselves and also preventing bugs from happening in our code because the function is never run when the path is the wrong type.

format	Accepts
string	(default) accepts any text without a slash
int	accepts positive integers
float	accepts positive floating point values
path	like string but also accepts slashes
uuid	accepts UUID strings

We can use this to more narrowly specify what path variable types are acceptable like so:

```
invoices = {  
    1: "invoice 1",  
    2: "invoice 2",  
}  
  
@app.route("/invoice_number/<num:int>/")  
def get_invoice(num: int):  
    """Returns the details for the given invoice number"""  
    invoice_info = invoices[num]  
    return f"Invoice number {num}: {invoice_info}"
```

Now if we attempt to access this endpoint with `/invoice_number/NotNumber` we will get a 404 error immediately.

27.2 Logging in Flask

Flask uses the built in Python logging library to handle logging. This is made accessible via `app.logger`.

The documentation states the following about setting up logging:

When you want to configure logging for your project, you should do it as soon as possible when the program starts. If `app.logger` is accessed before logging is configured, it will add a default handler. If possible, configure logging before creating the application object.

Since Flask is using the default Python logging library you can use those same general techniques you would have used outside Flask. Please refer to the general logging materials for more details.

One common pattern is to register a `logging.handlers.SMTPHandler` to automatically send an email to the website administration team when sufficiently critical errors are encountered. The pattern will look like this:

```
import logging
from logging.handlers import SMTPHandler

mail_handler = SMTPHandler(
    mailhost='127.0.0.1',
    fromaddr='[email protected]',
    toaddrs=['[email protected]'],
    subject='Application Error'
)
mail_handler.setLevel(logging.ERROR)
mail_handler.setFormatter(logging.Formatter(
    '[%(asctime)s] %(levelname)s in %(module)s: %(message)s'
))
if not app.debug:
    app.logger.addHandler(mail_handler)
```

Note that for this to work you must have the SMTP server set up on the same

machine that is running the Flask app.

27.3 Error handlers

One of the design decisions with Flask was the ability to use exceptions to represent HTTP error codes, this means that in many cases Exceptions are being directly used to control the flow of the web applications. Here's how we can use the `abort` function to represent a 404 error via the exception it will raise.

```
from flask import Flask, abort
app = Flask(__name__)
invoices = {
    1: "invoice 1",
    2: "invoice 2",
}

@app.route("/invoice_number/<num:int>/")
def get_invoice(num: int):
    """Returns the details for the given invoice number"""
    try:
        invoice_info = invoices[num]
    except KeyError:
        abort(404) # immediately raise a 404 exception
    return f"Invoice number {num}: {invoice_info}"
```

1

If one of these Flask HTTP exceptions propagates Flask will attempt to handle this by calling the error handler for that particular status code.

If you have an unhandled Python exception in your code the Flask framework will catch this exception and give a generic 500 error message to the user. If you are writing your own code it makes sense to hook into this system to do important things like logging.

Dealing with unhandled Python exceptions

Whenever we have an unhandled Python exception in our application we want to make sure that we know about it via logging and other means.

We can use `@app.errorhandler` to catch Python exceptions but since flask uses exceptions to indicate non-success HTTP codes we have to be careful with what we catch:

Since the Flask HTTP exceptions derive from `werkzeug.exceptions.HTTPException` we have to make sure we do *not* catch those:

```
from werkzeug.exceptions import HTTPException #base for all
    ↵ flask HTTP exceptions

@app.errorhandler(Exception)
def handle_exception(e):
    # pass through Flask Exceptions
    if isinstance(e, HTTPException):
        return e

    # All exceptions here are uncaught non-Flask exceptions
    # Log at some high severity here
    # Also perhaps roll back any DB transactions in progress
    ↵ here
    return render_template('500_generic.html', e=e), 500
```

Note that if we were serving an API we would generate something like a JSON response here.

27.4 Reversing URLs

When we are writing functions that will handle our URL endpoints we sometimes want to be able to find the URL that is being handled for a given function.

For example, let's say we have an endpoint like this:

```
from flask import Flask
app = Flask(__name__)
@app.route("/login")
def authenticate():
    return "This is the login page!"

@app.route("/")
def home():
    auth_url = "/login"
    return f"The auth page can be found at: {auth_url}:"
```

This particular way of generating the contents of `auth_url` is problematic because in the future that URL route could change. To deal with this in a reliable way Flask provides the `url_for` helper to find out what URL route is associated with a python function:

```
from flask import Flask, url_for

@app.route("/login")
def authenticate():
    return "This is the login page!"

@app.route("/")
def home():
    auth_url = url_for(authenticate)
    return f"The auth page can be found at: {auth_url}:"
```

Now the URL will always be properly generated, if we changed `/login` to `/authenticate` then the `/` route will always display the right URL to find our `authenticate` function.

Exercise: dynamically create a sitemap for the following

```
users = ["Ed", "Henry", "Errol", "Janis"]
products = [1,2,3,4]
```

```
@app.route("/products")
def products():
    """Return products information
    sitemap: product information
    """
    return products

@app.route("/users")
def users():
    """Return user information
    sitemap: user information
    """
    return users

@app.route("/login")
def authenticate():
    """
    Authenticate a user
    sitemap: This is the login page
    """
    return "This is the login page!"

@app.route("/")
def home():
    """Homepage
    sitemap: homepage
    """
    auth_url = url_for(authenticate)
    return f"The auth page can be found at: {auth_url}:"
```

- Hint: The function documentation can be accessed via `__doc__`
- Hint: you can access the names defined in the global state via `globals()`

Chapter 28

Network automation

28.1 SSH automation with Fabric

This section is adapted from the notes by Jason McVetta from here:

<https://github.com/jmcvetta/curriculum-advanced-python.git>. Licence: CC BY 3.0.

Fabric is a simple but powerful SSH automation tool that is widely used in industry. It streamlines the use of SSH for application deployment or systems administration tasks.

Fabric provides a basic suite of operations for executing local or remote shell commands (normally or via sudo) and uploading/downloading files, as well as auxiliary functionality such as prompting the running user for input, or aborting execution.

You can install fabric with `pip install fabric`

```
! fab --version
```

68

```
Fabric 2.3.1
Paramiko 2.4.1
Invoke 1.1.0
```

28.2 Fabric Connection

The most basic use of Fabric is to connect to a shell via a Connection, execute a command (via the Python Invoke library) and return the result.

```
import getpass
```

6

```
username = getpass.getuser()  
password = getpass.getpass('Please enter your password: ')
```

7

Please enter your password:

```
from fabric import Connection
```

3

```
Connection('localhost', user=username,  
          connect_kwargs={'password': password}).run('uname -s')
```

15

Darwin

```
<Result cmd='uname -s' exited=0>
```

Note that The response from Fabric by default will echo to the system standard out, as well as returning a Result object. Passing the hide keyword argument will stop the automatic echo, though the server response can be easily obtained from the result object.

```
result = Connection('localhost', user=username,  
                    connect_kwargs={'password': password}).run('uname -s',  
                    hide=True)
```

18

```
result.stdout
```

19

```
'Darwin\n'
```

A note on SSH

Fabric uses the underlying Python Paramiko library to connect via Secure Shell (SSH) to a given machine. On most Linux / macOS machines this is not too difficult to set up - you will likely have an OpenSSH service installed and enabled. On a Windows machine this can be a little more tricky as you will likely need Administrator privileges to install and configure the server.

As such for the purpose of training you have access to the following access details:

Username	trainee
Password	*****
Host	ec2-13-211-219-60.ap-southeast-2.compute.amazonaws.com

Ask us for the password in your training course.

```
conn = Connection(  
    'ec2-13-211-219-60.ap-southeast-'  
    '2.compute.amazonaws.com',  
    user='trainee',  
    connect_kwargs={'password': password}  
)
```

24

```
conn.run('cat success.txt')
```

25

```
Connection successful
```

```
<Result cmd='cat success.txt' exited=0>
```

Executing commands over a Connection

Once you have a Connection set up you can use Fabric to execute commands directly on the server. You've already seen the run command which will execute a shell command on the server and return the response:

```
conn.run('whoami')
```

28

```
trainee  
<Result cmd='whoami' exited=0>
```

Other commands are available:

- sudo allowing a user to use sudo permissions to execute a command on the server, assuming of course the user has permission
- cd changes directory - this can also be used as a context manager
- put allows a user to copy a file from the local system to the remote system

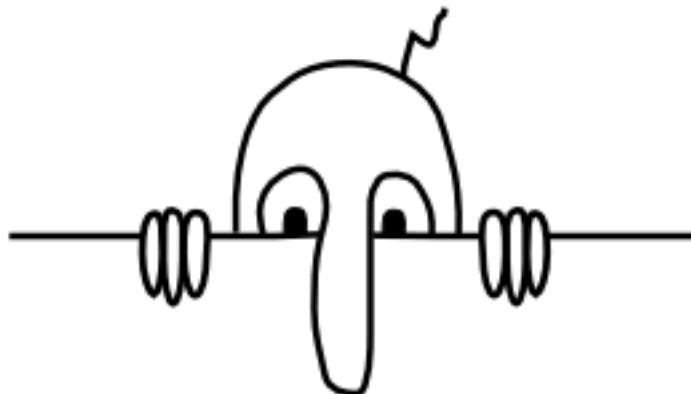
```
conn.put('Data/Auto.csv', remote='Auto.csv')  
conn.run('ls')
```

34

```
Auto.csv  
success.txt  
<Result cmd='ls' exited=0>
```

Exercise: Making your mark on an open server

Make your mark on an open server by leaving a sign if you were there. Connect to the open server above and copy across the file Data/kilroy_was_here.png to the server with your own name on the file.



Kilroy
was here

Run the `ls` command to guarantee that the file is there based on the `stdout`.

```
#%load solutions/kilroy.py
```

40

You can also copy files across from the server to your local machine with the `get` method:

```
conn.get('henry.png', 'local_file.png')
```

42

```
<fabric.transfer.Result at 0x1074d1c88>
```

```
import os
os.path.exists('local_file.png')
```

44

True

Writing commands on multiple servers

It's all well and good to connect to different machines individually and complete the same work:

```
for host in [
    'ec2-13-211-219-60.ap-southeast-'
    ↵ 2.compute.amazonaws.com',
    'ec2-13-211-219-60.ap-southeast-'
    ↵ 2.compute.amazonaws.com']:
    conn = Connection(
        host,
        user='trainee',
        connect_kwargs={'password': password})
    conn.run('uname -s')
```

Linux

Linux

That works well if you want to execute tasks in order. Instead it's more likely that you'll want to execute the same commands synchronously across different hosts. This is easily using a fabric Group. You can subclass the Group class yourself, but the easiest way to use groups is to use the built-in SerialGroup or the ThreadingGroup which execute in serial and parallel respectively.

```
from fabric import SerialGroup, ThreadingGroup
```

This is easy if you have the host strings set up appropriately using fabric configuration (see below), but if you have to pass extra arguments like the username and the password you can make a group from collections:

```
connections = []
for host in [
    'ec2-13-211-219-60.ap-southeast-'
    ↵ 2.compute.amazonaws.com',
    'ec2-13-211-219-60.ap-southeast-'
    ↵ 2.compute.amazonaws.com']:
    conn = Connection(
        host,
        user='trainee',
        connect_kwargs={'password': password})
    connections.append(conn)
```

48

```
results = ThreadingGroup.from_connections(
    ↵ connections=connections).run('uname
    ↵ -s')
```

54

Linux

Linux

The ResultsGroup that is returned is an extended dictionary keyed off the original connections, but with two important attributes:

```
results.succeeded
```

59

```
{<Connection host=ec2-13-211-219-60.ap-southeast-'
    ↵ 2.compute.amazonaws.com user=trainee>: <Result cmd='uname
    ↵ -s' exited=0>}
```

```
results.failed
```

60

```
{}
```

Which split results into successes and fails respectively.

28.3 Configuring fabric hosts

So you don't have to keep configuring long host names, usernames and passwords the best strategy is often to build configuration files to go with Fabric.

By default Fabric will read in your SSH configuration from: `~/.ssh/config`. Note that your SSH config will not support storage of a raw password, so for the example connection we have above you should add the following lines to `~/.ssh/config`:

```
Host python_trainee
    HostName ec2-13-211-219-60.ap-southeast-2.compute.amazonaws.com
    Port 22
    User trainee
```

Then from the command line run

```
ssh-copy-id python_trainee
```

After entering your password your computer's identity will be recorded on the remote machine and you won't need to record a password the next time you log into the machine.

```
! fab -H python_trainee hostname
```

76

```
ip-172-31-5-84
```

You can also override configuration in the the Fabric system, based on the Invoke library configuration hierarchy:

1. Internal default values
2. Collection-driven configuration
3. System level configuration, stored in `/etc/fabric.yml`
4. User-level configuration file in `~/.fabric.yml`
5. Project-level configuration file in the current project in `fabric.yml`
6. Via Environment variables
7. Runtime configuration file whose path is given to `-f`, e.g. `fab -f /random/path/to/config_file.yaml`.
8. Command-line flags for certain core settings, such as `-e`.

9. Modifications made by user code at runtime.

28.4 fab Command Line Interface

Fabric provides a command line utility `fab`, which reads its configuration from a file named `fabfile.py` in the directory from which it is run. A typical fabfile contains one or more functions to be executed on a group of remote hosts.

The following example provides functions to check free disk space and host type, as well as defining a group of hosts on which to run:

```
%%writefile fabfile.py
from fabric import task

@task
def hosttype(conn):
    conn.run('uname -s')

@task
def hostname(conn):
    conn.run('hostname')
```

80

Overwriting `fabfile.py`

Fabric can be invoked from the command line with the `fab` command, which by default will look for functions decorated with tasks in `fabfile.py`. For example:

```
! fab --list
```

82

Available tasks:

```
hostname
hosttype
```

Once a task is defined, it may be run on one or more servers, like so:

```
! fab -H python_trainee hosttype
```

81

Linux

Task arguments

It's often useful to pass runtime parameters into your tasks, just as you might during regular Python programming. Fabric has basic support for this using a shell-compatible notation: <task name> <arg> --<kwarg>=<value>, It's contrived, but let's extend the above example to say hello to you personally:

```
%%writefile fabfile.py
from fabric import task

@task
def hello(c, name="world"):
    print("Hello %s!" % name)
```

94

Overwriting fabfile.py

By default, calling `fab hello` will still run and print “Hello world!”, but you can also pass through command line arguments:

```
! fab hello --name Henry
```

98

Hello Henry!

For now, your argument values will always show up in Python as strings and may require a bit of string manipulation for compound types such as lists. Future versions may add a typecasting system to make this easier.

Library Usage

In addition to use via the `fab` tool, Fabric's components may be imported into other Python code, providing a Pythonic interface to the SSH protocol suite at

a higher level than that provided by e.g. the `ssh` library (which Fabric itself uses.)

Exercise: Current IP address

Consider the case where we want to collect the IP address from a group of hosts.

1. Write a script that uses the Fabric library to poll a group of hosts for their IP addresses, parses it and displays the IP address to the user:

Use this as a template:

```
from fabric import task

@task
def get_ip_address(conn):
    result = conn.run('ip addr show', hide=True)

    # write your parsing code here
```

1

Test it on the `python_trainee` host above.

28.5 More libraries for network automation

- `telnetlib`: part of the official Python standard library: <https://docs.python.org/3.6/library/telnetlib.html>
- Ansible
- Paramiko
- Netmiko
- PySNMP
- Juniper's PyEZ library
- Arista's eAPI
- NAPALM

Chapter 29

Regular expressions

In this exercise, we will learn about parsing a text file using regular expressions. We will analyse a Cisco log file using the `re` module and extract log messages from the file.

29.1 Regular expressions and the `re` module

Regular expressions are a powerful way to search, replace and parse complex text.

They can also be cryptic, tricky and error-prone.

However, used well, they are more readable than a long chain of string functions. They also tend to be implemented more efficiently by optimised packages (such as Python's `re` module).

A good introduction to regular expressions with Python can be found here:

http://www.diveintopython.net/regular_expressions/index.html

A more comprehensive tutorial for regular expressions is here:

<http://regex.learncodethehardway.org/book/>

For this exercise, you need to remember the following basics of regular expressions. More details will be given in the following exercises.

^ matches the beginning of a string.
\$ matches the end of a string.
. matches any character.
\b matches a word boundary.
\d matches any numeric digit.
\D matches any non-numeric character.
\s matches any whitespace character (tab, newline, etc.)
\S matches any non-whitespace character
\w matches any alphanumeric character (letters and numbers)
x? matches an optional x character (in other words, it matches zero or one x).
x* matches x zero or more times.
x+ matches x one or more times.
x{m,n} matches an x character at least n times, but not more than m.
[a|b|c] matches exactly one of a, b or c.
(x) in general is a remembered group. You can get the value of x.

The `re` module is the standard module for regular expression processing with Python. You can find the details of the module at <https://docs.python.org/3/howto/regex.html>.

Extract a Cisco IOS version number.

Throughout this session, we will analyse Cisco log messages in the `show_tech.log` file.

Let's extract a version number of the Cisco IOS software from the log file.

Here is a string example to parse.

```
message = 'Version 15.4(20141009:175959)'
```

2

```
import re
pattern = r"""
    Version      # start with Version
    \s+          # white space(s). \s for a white space, + for
    ↵ one or more times
    (\d+\.\d+)   # the version number, xxx.xxx. () to create a
    ↵ group. \d for any numeric digit.
```

3

```
\(          # left parenthesis
(\d+):(\d+) # build numbers, xxxx:xxxx
\)
"""
parser = re.compile(pattern, re.VERBOSE) # re.VERBOSE with
    ↵ verbose expression
searched = parser.search(message)
searched.groups()
```

('15.4', '20141009', '175959')

Please note that using (x) allows you to extract values by calling groups(). If you want to include a parenthesis in a search pattern, prefix with backslash like \ (and \).

You can use a compact regular expression to define a pattern:

```
pattern = r'Version\s+(\d+\.\d+)\((\d+):(\d+)\)'
parser = re.compile(pattern)
searched = parser.search(message)
searched.groups()
```

4

('15.4', '20141009', '175959')

Make sure to use a raw string prefixed with r when you use a compact expression to prevent accidental conversion of escape characters e.g. '\t' to a tab.

In this tutorial, we will use only verbose expressions for didactic purposes. With additional comments, the verbose expression allows you to put more details to each part of a complex pattern.

“Readability counts”

(Source: The Zen of Python at <https://www.python.org/dev/peps/pep-0020/>)

Unlike compact expressions, in a verbose regular expression,

- Whitespace is ignored. Spaces, tabs, and carriage returns are not matched. Instead use \s for white space matching.

- Comments are ignored.

Don't forget to use the `re.VERBOSE` flag as above when compiling a verbose regular expression.

Now, using the parser, you can exact a version number of the Cisco IOS Software directly from the following header in the log file.

```
Cisco IOS XE Software, Version 2014-10-28_16.35_vipbansa  
Cisco IOS Software, ASR1000 Software  
↳ (X86_64_LINUX_IOSD-ADVIPSERVICESK9-M), Experimental  
↳ Version 15.4(20141009:175959) [v154_3_s_xe313_throttle-]  
↳ vipbansa-V154_3_S_FC11-with-CSCun20543-removal  
↳ 111]  
Copyright (c) 1986-2014 by Cisco Systems, Inc.  
Compiled Tue 28-Oct-14 15:24 by vipbansa
```

```
# Read the entire text and parse it with the pattern.  
parser.search(open("/data/net/show_tech.log").read()  
↳ ).groups()
```

```
('15.4', '20141009', '175959')
```

Exercise

The `show_tech.log` file also has a version number for the Cisco IOS XE Software. Create a regular expression pattern and extract the version number from the log file.

```
!head -n50 /data/net/show_tech.log | tail -n6
```

Hint: to match the username, use `\w` to match any alphanumeric character (not a newline).

```
import re
pattern_xe = """
    Version      # start with Version
    \s+          # white space(s). \s for a white space, + for
    ↵ one or more times
    (\d+-\d+-\d+)  # the compile date in ISO 9660 format:
    ↵ YYYY-mm-dd
                    # () to create a group. \d for any
    ↵ numeric digit.

    -
    (\d+)\.(\d+) # other version numbers

    -
    (\w+)        # username who has compiled it (one or more
    ↵ letters)
"""

```

```
message = open("/data/net/show_tech.log").read()
parser_xe = re.compile(pattern_xe, re.VERBOSE | re.MULTILINE)
searched_xe = parser_xe.search(message)
searched_xe.groups()
```

('2014-10-28', '16', '35', 'vipbansa')

Exercise (b)

Try this example with \D instead of \w to match the username, to see the difference. Read up on the `re.MULTILINE` regex compiler flag in the Python “Regular Expressions HOWTO”.

29.2 Extract lines with specific log messages

The log file has several log messages that look like the following string.

```
message = "*Jan  6 21:47:16.780: %LINEPROTO-5-UPDOWN: Line
    ↵ protocol on Interface TenGigabitEthernet0/1/0, changed
    ↵ state to up"
```

The first part of the log message, 'Jan 6 21:47:16.780', is the recorded date and time of the log message.

The second part of the log message, %LINEPROTO-5-UPDOWN, is the message code and severity level. In this message, the message code family is LINEPROTO, the priority level is 5, and a family type of UPDOWN.

Each log message is categorised by the following severity types: 0 for the most critical message and 7 for the least severe one.

ACS Severity Level	Description	Syslog Severity Level
FATAL	Emergency. ACS is not usable and you must take action immediately.	1 (highest)
ERROR	Critical or error conditions.	3
WARN	Normal, but significant condition.	4
NOTICE	Audit and accounting messages. Messages of severity NOTICE are always sent to the configured log targets and are not filtered, regardless of the specified severity threshold.	5
INFO	Diagnostic informational message.	6
DEBUG	Diagnostic message.	7

So, LINEPROTO-5-UPDOWN is a simple notice.

And the final part is the message body, the human readable text of each log message.

Using the following pattern, you can extract the message code from the message.

```
pattern = """%
  (          # defining a group
    [A-Z]+   # message class family
    \-        # - delimiter
    \d+       # severity code
    \-        # - delimiter
    [A-Z]+   # family type
  )          # closing a group
"""

parser = re.compile(pattern, re.VERBOSE)
searched = parser.search(message)
```

44

```
searched.groups()
```

('LINEPROTO-5-UPDOWN',)

Now with the parser, let's extract all error codes in the log file. `findall()` will find all non-overlapping occurrences.

```
parser.findall(open("show_tech.log").read())
```

34

```
[ 'CLNS-4-DUPSNPA',
  'LINK-3-UPDOWN',
  'LINK-3-UPDOWN',
  'LINEPROTO-5-UPDOWN',
  'LINK-3-UPDOWN',
  'LINK-3-UPDOWN',
  'LINEPROTO-5-UPDOWN',
  'LINEPROTO-5-UPDOWN',
  'LINK-3-UPDOWN',
  'LINEPROTO-5-UPDOWN',
  'SYS-5-CONFIG']
```

You can also extract all lines with log messages in this format.

```
for line in open("show_tech.log"):
    if parser.search(line): # note that None is treated as
        False in Python.
        print(line)
```

46

```
*Jan  2 22:03:35.210: %CLNS-4-DUPSNPA: ISIS: Duplicate SNPA
  ↵  0024.976a.62bf detected

*Jan  3 03:27:45.444: %LINK-3-UPDOWN: Interface
  ↵  TenGigabitEthernet0/1/0, changed state to down

*Jan  3 03:27:45.443: %LINK-3-UPDOWN: SIP0/1: Interface
  ↵  TenGigabitEthernet0/1/0, changed state to down
```

```
*Jan  3 03:27:46.445: %LINEPROTO-5-UPDOWN: Line protocol on
  ↵  Interface TenGigabitEthernet0/1/0, changed state to down

*Jan  6 21:47:15.781: %LINK-3-UPDOWN: Interface
  ↵  TenGigabitEthernet0/1/0, changed state to up

*Jan  6 21:47:15.779: %LINK-3-UPDOWN: SIP0/1: Interface
  ↵  TenGigabitEthernet0/1/0, changed state to up

*Jan  6 21:47:16.780: %LINEPROTO-5-UPDOWN: Line protocol on
  ↵  Interface TenGigabitEthernet0/1/0, changed state to up

*Jan  6 22:20:16.653: %LINEPROTO-5-UPDOWN: Line protocol on
  ↵  Interface Port-channel20, changed state to down

*Jan  6 22:22:48.414: %LINK-3-UPDOWN: Interface
  ↵  Port-channel20, changed state to up

*Jan  6 22:22:49.413: %LINEPROTO-5-UPDOWN: Line protocol on
  ↵  Interface Port-channel20, changed state to up

*Jan  6 22:23:54.461: %SYS-5-CONFIG_I: Configured from console
  ↵  by console
```

Now, let's extract each part of a log message line. Here is a string to test our patterns.

```
message = "*Jan  6 21:47:16.780: %LINEPROTO-5-UPDOWN: Line
  ↵  protocol on Interface TenGigabitEthernet0/1/0, changed
  ↵  state to up"
```

47

```
pattern = """
^          # beginning of a string
\*          # start with *
(.*)
until      # the rest of the expression matches
:"
```

123

```
\s*
%
(
    # message code
[A-Z] +
\-
\d+
\-
[A-Z] +
)
:
\s*
(.*)
"""
parser = re.compile(pattern, re.VERBOSE)
searched = parser.search(message)
searched.groups()
```

```
('Jan  6 21:47:16.780',
'LINEPROTO-5-UPDOWN',
'Line protocol on Interface TenGigabitEthernet0/1/0, changed
↳ state to up')
```

The parser returns three components of the log message as a tuple. We assign each component to a temporary variable for further parsing.

```
the_date, the_code, the_body = searched.groups()
the_date, the_code, the_body
```

124

```
('Jan  6 21:47:16.780',
'LINEPROTO-5-UPDOWN',
'Line protocol on Interface TenGigabitEthernet0/1/0, changed
↳ state to up')
```

Exercise

The `show_tech.log` has the following messages about redundancy history. Create a regular expression pattern and parse each line with `RF_STATUS_REDUNDANCY_MODE_CHANGE`.

For instance, return ('00:16:39', 'RF_STATUS_REDUNDANCY_MODE_CHANGE(405) LACP(226) op=7 rc=0')

by parsing the following line:

```
00:16:39 RF_STATUS_REDUNDANCY_MODE_CHANGE(405) LACP(226) op=7  
    ↵   rc=0
```

1

Parsing each component of a log message further

Let's parse the first part, the date and time when each message was logged.

```
the_date
```

52

```
'Jan  6 21:47:16.780'
```

```
pattern = """  
    ^          # beginning of a string  
    (\D{3})    # month  
    \s+        # white space(s)  
    (\d+)      # day  
    \s+        # white space(s)  
    (\d+)      # hour  
    :          # delimiter :  
    (\d+)      # minute  
    :          # delimiter :  
    (\d+)      # second  
    \.          # floating point  
    (\d+)      # a fraction of a second  
    $          # end of a string  
"""  
  
parser = re.compile(pattern, re.VERBOSE)  
searched = parser.search(the_date)  
searched.groups()
```

54

```
('Jan', '6', '21', '47', '16', '780')
```

29.3 Parsing dates and times

Often we may wish to convert this to a Python `datetime` object instead of these strings. We can do this as follows:

```
import datetime
```

58

```
# See the docs for the `datetime.strptime()` function  
# for the meaning of the codes below:  
date_format = "%b %d %H:%M:%S.%f"
```

89

```
the_date
```

90

```
'Jan 6 21:47:16.780'
```

```
my_date = datetime.datetime.strptime(the_date, date_format)
```

107

```
my_date
```

108

```
datetime.datetime(1900, 1, 6, 21, 47, 16, 780000)
```

Notice that this assumes the year 1900. To replace it with the current year, we can use:

```
my_year = datetime.date.today().year
```

117

```
my_date.replace(my_year)
```

118

```
datetime.datetime(2015, 1, 6, 21, 47, 16, 780000)
```

If specifying the date/time format precisely like this seems too laborious, you can use the 3rd-party `dateutil` package:

```
!pip install python-dateutil
```

95

```
Requirement already satisfied (use --upgrade to upgrade):  
  - python-dateutil in  
    /Users/schofield/anaconda/lib/python3.4/site-packages  
Cleaning up...
```

Among its features, it offers “generic parsing of dates in almost any string format”:

```
from dateutil.parser import parse
```

119

```
parse(the_date)
```

120

```
datetime.datetime(2015, 1, 6, 21, 47, 16, 780000)
```

Now, we also break down the message code into three components.

```
the_code
```

121

```
'LINEPROTO-5-UPDOWN'
```

```
pattern = """  
^          # beginning of a string  
([A-Z]+)   # message code family  
-          # delimiter -  
(\d+)      # severity  
-          # delimiter -  
([A-Z]+)   # family type  
$          # end of a string  
"""
```

126

```
parser = re.compile(pattern, re.VERBOSE)  
searched = parser.search(the_code)  
searched.groups()
```

```
('LINEPROTO', '5', 'UPDOWN')
```

Exercise

Extract the time and message parameters from each RF_STATUS_REDUNDANCY_MODE_CHANGE message.

For instance, return the following tuple:

```
('00', '16', '39', 'RF_STATUS_REDUNDANCY_MODE_CHANGE',
 ↵ '405', 'LACP', '226', 'op', '7', 'rc', '0')
```

from the following string:

```
'00:16:39 RF_STATUS_REDUNDANCY_MODE_CHANGE(405) LACP(226)
↪ op=7 rc=0'
```

Extended Exercise

In the data folder, one of the files is a private key. Using a regular expression and `os.walk` or `glob`, find that file.

```
# See solutions/private_key.py"
```

29.4 Other resources

- <https://medium.com/@youknowjamest/parsing-file-names-using-regular-expressions-3e85d64deb69>
- Understanding Regular Expressions (12 minute video): <https://youtu.be/DRR9fOXfRE>
- Visualize Regular Expressions: <https://regexr.com/>
- Interactive Tutorial to learn Regular Expressions: <https://regex-one.com/>
- Beginners Tutorial about Regular Expressions: <https://www.analyticsvidhya.com/blog/2015/06/regular-expression-python/>

Chapter 30

Parsing semi-structured data

30.1 Intro to TextFSM

The TextFSM library is a parsing library that allows you to conveniently parse semi-structured data.

When dealing with various network and system admin tasks with things that do NOT have programmatic APIs you'll often find that you get structured text outputs that you want to parse. This library is useful for parsing such data.

Install with:

```
pip install textfsm
```

1

The FSM in the name TextFSM refers to a Finite State Machine, this is a conceptual model of computation that the TextFSM library is built on top of.

The overall goal

The goal when using `textfsm` is to get the information from devices into our Python scripts in such a way that we can easily automate things. We want to be able to get information like this:

Interface	Status	Protocol Description
-----------	--------	----------------------

Fa1	admin	down	down
Te1/1	up	up	lexample/Eth2/22
Te1/2	up	up	oexample/Eth2/22

into Python like this:

```
['Fa1','admin down','down','']  
['Te1/1','up','up','lexample/Eth2/22']  
['Te1/2','up','up','oexample/Eth2/22']
```

1

When you might want to use this library

If you have some complicated tabular data without an API to interact with it, you may consider parsing the textual output directly.

If you have some tabular data that has some information in it that you wish to extract, as long as only one entry per row matches a specific textual format you could go about this task in many ways. One such way is to use a regular expression. For some simple tasks this gets the job done, but there are issues that come up with more complex tasks.

One of the most annoying situations is when you have free-form comments or rows with optional output since this can make parsing considerably harder. We can see some of these issues in the following example:

```
! python3 extras/phone_project.py
```

1

number	status	project	description
0491 570 157	active	A	used to be in project B
0491 570 158	active	A	
0491 570 159	disconnected	A	
0491 570 110	active	B	was disconnected on → 2019/12/01 activated again 2019/12/02
0491 570 313	active	B	previously was 0491 573 087
0491 570 737	inactive	B	Deactivated 2019/06/09
0491 571 266	disconnected	C	
0491 571 491	disconnected	C	

```
0491 571 804 active      C
0491 572 549 active      C
0491 572 665 active      Update
0491 572 983 inactive    Update  Deactivated 2019/06/01
0491 573 770 disconnected Update
0491 573 087 disconnected
```

A non-contrived example is the Cisco show interfaces description where one of the output columns is a free form text description that people can enter into the devices.

You could try to parse some text like this by splitting each row based on whitespace and positions. For example here you have the option to split each row when you see two or more spaces. Having that free-form “description” column can make this hard to do depending on what is in it, as a user could enter in more than one space by accident for example.

Or you could put together a regex to search for particular terms. This would have the downside of perhaps matching things in the wrong column. Once again this description column can be annoying; we would need to have some logic to make sure that if we matched a value for a phone number it would only be at the start of the line. Including this context starts to make the regex more complicated to write and more prone to errors.

TextFSM allows you to nicely combine the good parts of both the regex approach and the positional parsing, which gives you substantially more power to parse the text.

For tabular structured text data TextFSM is often more convenient than having to roll out your own fully customized parser using something like PyParsing or LR parser or some other parser generator.

30.2 TextFSM templates

To use TextFSM you will need to first create a template that defines the structure of the expected output.

It is by defining this structure that you get the power from a parser like TextFSM.

A TextFSM template consists of:

1. Value Definitions
2. State Definitions
3. Actions

Values are the smallest building block and will allow you to specify a value you may find in a record, this is similar to defining the contents of a cell in a tabular data format. States are a groups of values according to a specific format, these specify valid rows. Actions lets you define what happens when you find a match of a State.

Value definitions

These specify what can be in an entry of data, usually representing a column, these go at the top of a template.

To define a value we have to use the `Value` keyword, followed by the name, followed by a regex that must be matched for extracting that column.

For example:

```
Value PORT (\S+)
Value STATUS (up|down|admin\s+down)
Value PROTOCOL (up|down)
Value DESCRIPT (.*?)
```

Values are like a cell in a tabular data format. Much like regex capture groups each value is used later to map to an item in the parsed output.

State definitions

A state is effectively a way of defining what makes up an item which you want to extract.

To define a state we have to create a pattern that will match the state. Because we are dealing with textual input, a state can be spread across more than one row of text.

We define a rule per line of text for each state:

```
stateName
```

```
^rule1  
^rule2  
...  
^ruleN
```

Where each rule is comprised of a pattern that includes whatever combination of regexes and Values we need.

Note: each line under a state name must start with exactly 2 spaces!

Actions

Once you have defined Values and how they combine into States you have to specify how they transfer to other states.

There are some special predefined Actions that will do useful things:

- Record will store this state in a results record
- Error will bail out the whole parsing process and discard all results so far

When you have a match for a state you then have to decide how to handle that match, this is what TextFSM refers to as an action.

The most common Action we will use is Record which tells TextFSM to store a match in a record. The next most common Action is to transfer to another State; this is useful for some more complex parsing tasks where you may have something like nested data.

30.3 Example: a simple template

Let's parse the following text output into a nice Python data structure:

```
! python3 extras/phone_project.py --simple
```

2

```
number      status  
0491 570 157 active  
0491 570 158 active  
0491 570 159 disconnected
```

```
0491 570 110 active
0491 570 313 active
0491 570 737 inactive
0491 571 266 disconnected
0491 571 491 disconnected
0491 571 804 active
0491 572 549 active
0491 572 665 active
0491 572 983 inactive
0491 573 770 disconnected
0491 573 087 disconnected
```

Note that there is one record per line in the data being output here.

```
%%writefile phone_simple.textfsm
Value PhoneNumber (\d+ \d+ \d+)
Value Status (\S+)

Start
^${PhoneNumber}\s+${Status} -> Record
```

Overwriting phone_simple.textfsm

```
raw_text_data = """number      status
0491 570 157 active
0491 570 158 active
0491 570 159 disconnected
0491 570 110 active
0491 570 313 active
0491 570 737 inactive
0491 571 266 disconnected
0491 571 491 disconnected
0491 571 804 active
0491 572 549 active
0491 572 665 active
0491 572 983 inactive
0491 573 770 disconnected
0491 573 087 disconnected"""

```

```
import textfsm

# Load in the TextFSM template
with open("extras/phone_simple.textfsm", mode='rt') as
    template:
        re_table = textfsm.TextFSM(template)
fsm_results = re_table.ParseText(raw_text_data)
fsm_results
```

```
[['0491 570 157', 'active'],
 ['0491 570 158', 'active'],
 ['0491 570 159', 'disconnected'],
 ['0491 570 110', 'active'],
 ['0491 570 313', 'active'],
 ['0491 570 737', 'inactive'],
 ['0491 571 266', 'disconnected'],
 ['0491 571 491', 'disconnected'],
 ['0491 571 804', 'active'],
 ['0491 572 549', 'active'],
 ['0491 572 665', 'active'],
 ['0491 572 983', 'inactive'],
 ['0491 573 770', 'disconnected'],
 ['0491 573 087', 'disconnected']]
```

A potential gotcha with matching end of lines

One thing you may have been tempted to do in the last example is to create a state that has to match the end of the row using \$\$, for example:

Start

```
^${PhoneNumber}\s+${Status}$$ -> Record
```

However, this can cause problems since trailing whitespace after the end of the shorter rows will NOT match if we explicitly state the Status ends the row exactly. Keep trailing whitespace in mind if you get missing matches when using \$\$ for ends of row matching.

There are times where \$\$ is very useful. For example, with free-form entries

like descriptions it can be very good to match the end of the line explicitly, since these columns can be arbitrary in length and contents but always end at the end of the line.

30.4 Exercise: parsing semi-structured text

Create a template to parse the original example text table:

```
!python3 extras/phone_project.py
```

5

number	status	project	description
0491 570 157	active	A	used to be in project B
0491 570 158	active	A	
0491 570 159	disconnected	A	
0491 570 110	active	B	was disconnected on → 2019/12/01 activated again 2019/12/02
0491 570 313	active	B	previously was 0491 573 087
0491 570 737	inactive	B	Deactivated 2019/06/09
0491 571 266	disconnected	C	
0491 571 491	disconnected	C	
0491 571 804	active	C	
0491 572 549	active	C	
0491 572 665	active	Update	
0491 572 983	inactive	Update	Deactivated 2019/06/01
0491 573 770	disconnected	Update	
0491 573 087	disconnected		

We have 4 columns so we need 4 Value definitions. The first 3 are required and the last is optional.

Create a TextFSM template that will extract the data from this formatted output.

Hint the free-form description column can use a regex like `(\S.*?)`

```
# See solutions/parse_phone_projects_table.py
```

1

30.5 Storing output in a Pandas DataFrame

If you want to do some sort of analysis on these outputs it can be useful to store the results in a Pandas DataFrame.

To do this we have to create new DataFrame with the right column names then append to it as the rows come in from TextFSM.

```
raw_data_all = """number           status      project
                   description
0491 570 157   active       A      used to be in project
                   ↵ B
0491 570 158   active       A
0491 570 159   disconnected A
0491 570 110   active       B      was disconnected on
                   ↵ 2019/12/01 activated again 2019/12/02
0491 570 313   active       B      previously was 0491
                   ↵ 573 087
0491 570 737   active       B
0491 571 266   disconnected C
"""
# Load in the TextFSM template
with open("solutions/phone_info_all.textfsm", mode='rt') as
    ↵ template:
    re_table = textfsm.TextFSM(template)
fsm_results = re_table.ParseText(raw_data_all)

import pandas as pd
results = pd.DataFrame(columns=['number', 'status', 'project',
                                ↵ 'description'])

# Extract each row from the parsed input and load into
    ↵ DataFrame
for row in fsm_results:
    results = results.append({
        'number': row[0],
        'status': row[1],
        'project': row[2],
```

```
'description': row[3],  
}, ignore_index=True)  
  
results
```

	number	status	project	description
0	0491 570 157	active	A	used to be in project B
1	0491 570 158	active	A	
2	0491 570 159	disconnected	A	
3	0491 570 110	active	B	was disconnected on 2019/12/01 activated again...
4	0491 570 313	active	B	previously was 0491 573 087
5	0491 570 737	active	B	
6	0491 571 266	disconnected	C	

30.6 Example matching multiple lines of text

Using the phone project script again we can have a look at a different type of text format by using the --project option:

```
! python3 extras/phone_project.py --projects
```

12

Project: A

```
-----  
Manager : Ed  
Description: Project A
```

Project: B

```
-----  
Manager : Errol  
Description: Project B
```

Project: C

```
-----  
Manager : Henry  
Description: Project C
```

Project: Update

Manager : Janis

Description: Updating equipment using automation

This is a bit more complex than the example from before where we had only one line at a time in a record. TextFSM will let us parse this, we just need a state with multiple lines.

Lets start by defining the values we need:

- Project: One of “A”, “B”, “C” or “Update”
- Manager: String
- Description: String that can contain *anything*

Value ProjectName (A|B|C|Update)

Value ManagerName (\S+)

Value Description (\S.*?)

Now we have to define a state:

Start

```
^Project:\s ${ProjectName}$$  
^-\+$  
^Manager\s+: \s+${ManagerName}$$  
^Description\s+: \s+${Description}$$ -> Record
```

Running this

First we have to save our template file

```
%%writefile phone_projects.textfsm  
Value ProjectName (A|B|C|Update)  
Value ManagerName (\S+)  
Value Description (\S.*?)
```

Start

```
^Project:\s+${ProjectName}$$  
^-\+$
```

13

```
^Manager\s+: \s+\$\{ManagerName}\$\$  
^Description:\s+\$\{Description}\$\$ -> Record
```

Writing phone_projects.textfsm

```
import textfsm  
  
raw_text_data = """Project: A  
-----  
Manager : Ed  
Description: Project A  
  
Project: B  
-----  
Manager : Errol  
Description: Project B  
  
Project: C  
-----  
Manager : Henry  
Description: Project C  
  
Project: Update  
-----  
Manager : Janis  
Description: Updating equipment using automation  
"""  
  
# Load in the TextFSM template  
with open("phone_projects.textfsm", mode='rt') as template:  
    re_table = textfsm.TextFSM(template)  
fsm_results = re_table.ParseText(raw_text_data)  
fsm_results
```

```
[['A', 'Ed', 'Project A'],  
 ['B', 'Errol', 'Project B'],  
 ['C', 'Henry', 'Project C'],
```

```
['Update', 'Janis', 'Updating equipment using automation']]
```

30.7 Practical examples of TextFSM templates

Many common router commands have templates that you can use with them.

Many of these can be found here for common network device commands:

<https://github.com/networktocode/ntc-templates/>

Some packages like Netmiko will allow you to send commands and will parse the results for you based on which command is used. In the background this is using TextFSM to do the parsing.

Exercise: Parse output from the Cisco show interfaces description command

- Parse the data in found in `extras/show_interfaces_output.txt`
- How many are currently “admin down”?
- Extension: Find all the interfaces which contain “upgrade project” in the description

Use the following TextFSM template (found from the ntc-templates project):

```
%>%writefile
  ↵  extras/cisco_iso_show_interfaces_description.textfsm
Value PORT (\S+)
Value STATUS (up|down|admin\s+down)
Value PROTOCOL (up|down)
Value DESCRIPT (\\S.*?)

Start
  ^Interface\\s+Status\\s+Protocol\\s+Description\\s*$$ -> Begin
  ^\\s*$$
  # Capture time-stamp if vty line has command time-stamping
  ↵ turned on
  ^Load\\s+for\\s+
  ^Time\\s+source\\s+is
  ^. -> Error
```

3

```

Begin
  ^ ${PORT}\s+${STATUS}\s+${PROTOCOL}(?:\s+${DESCRIP})?\s*$$
    ↵ -> Record
  ^ \s*$$
    ↵ . -> Error

```

Writing `extras/cisco_iso_show_interfaces_description.textfsm`

```
# %load solutions/parse_cisco_show_interfaces_pandas.py
```

4

30.8 Example: analyzing and joining results

A task that will frequently come up is to pull data from multiple devices and then join those results such that you can do some analysis. Let's look at a way you can do this using Pandas.

In the example materials from before you can see there is a unique ID in the phone number that is occurring in both data sets. If we wanted to ask what manager was responsible for a phone number we would have to join the results from both the outputs.

```

import subprocess

raw_data_bytes = subprocess.check_output(
    ['python', 'extras/phone_project.py'])
summary_raw_data = raw_data_bytes.decode('UTF8')
print(summary_raw_data[:300])

```

18

number	status	project	description
0491 570 157	active	A	used to be in project B
0491 570 158	active	A	
0491 570 159	disconnected	A	
0491 570 110	active	B	was disconnected on
↪	2019/12/01	activated again	2019/12/02

```
0491 570 313 active          B
```

```
raw_data_bytes = subprocess.check_output(  
    ['python', 'extras/phone_project.py', '--projects'])  
projects_raw_data = raw_data_bytes.decode('UTF8')  
print(projects_raw_data[:155])
```

19

Project: A

Manager : Ed

Description: Project A

Project: B

Manager : Errol

Description: Project B

Create the Pandas DataFrames

```
with open("solutions/phone_info_all.textfsm", mode='rt') as  
    template:  
        re_table = textfsm.TextFSM(template)  
  
fsm_results = re_table.ParseText(summary_raw_data)  
results_table_data = pd.DataFrame(  
    fsm_results,  
    columns=['number', 'status', 'project', 'description'])  
  
results_table_data.head()
```

25

	number	status	project	description
0	0491 570 157	active	A	used to be in project B
1	0491 570 158	active	A	
2	0491 570 159	disconnected	A	
3	0491 570 110	active	B	was disconnected on 2019/12/01 activated again...
4	0491 570 313	active	B	previously was 0491 573 087

```
with open("extras/phone_projects.textfsm", mode='rt') as
    template:
        re_table = textfsm.TextFSM(template)

project_results = re_table.ParseText(projects_raw_data)
projects = pd.DataFrame(
    project_results,
    columns=['project', 'manager', 'description']
)
projects.head()
```

	project	manager	description
0	A	Ed	Project A
1	B	Errol	Project B
2	C	Henry	Project C
3	Update	Janis	Updating equipment using automation

Merging the DataFrames

You can merge the two dataframes using the Pandas `join` method or its more flexible cousin, `merge`:

```
merged = results_table_data.merge(
    projects,
    left_on="project", right_on='project')
```

```
merged.head()
```

28

	number	status	project	description_x
0	0491 570 157	active	A	used to be in project B
1	0491 570 158	active	A	
2	0491 570 159	disconnected	A	
3	0491 570 110	active	B	was disconnected on 2019/12/01 activated again...
4	0491 570 313	active	B	previously was 0491 573 087

	manager	description_y
0	Ed	Project A
1	Ed	Project A
2	Ed	Project A
3	Errol	Project B
4	Errol	Project B

Chapter 31

Further resources

31.1 Python Charmers materials

You will receive either a USB stick or access to download online materials containing Jupyter notebooks with the course content, solutions to the exercises, Python installers for multiple platforms, and reference documentation on Python and the third-party packages covered in the course.

Included documentation

An extensive set of free documentation, including reference manuals and several written tutorials, is provided as PDF files, either:

- on the USB stick in the Documentation folder
- or at github.com/PythonCharmers/PythonCharmersDocs

These are all under free licences. The individual licences are described in the **Documentation Licences.pdf** file in the Documentation folder on the USB stick.

31.2 Online Python resources

General Python resources

Stack Overflow - the top site for answers to programming questions, with a database of millions of questions. We highly recommend going through the most highly rated Python questions, e.g.: - Hidden Features of Python - Python Progression Path - From Apprentice to Guru.

The **Python Tutor** mailing list for basic questions about Python and programming

Mailing lists of third-party Python packages. You can often interact directly with the developers!

The **Python Package Index** (pypi.python.org): 200,000+ 3rd-party Python packages, and growing.

Python database resources

Wiki page for Python integration with common databases: <https://wiki.python.org/moin/DatabaseInterfaces>.

SQLAlchemy: a powerful object-relational mapper and database toolkit for Python

Scientific Python resources

See the **online videos** of the conferences **SciPy**, **EuroSciPy**, **PyData** (several), and **PyCon** (including regional ones) for up-to-date information.

Pint and **AstroPy** packages for easily handling scientific quantities and unit conversions

Scientific Python Lecture Notes at <http://scipy-lectures.org> (originally developed for the EuroSciPy conference)

SciPy topical software: <http://www.scipy.org/topical-software.html>. This is a human-curated list of packages sorted by scientific / technical field.

Python for data analysis and modelling

Pandas for powerful and convenient analysis of tabular data and time-series

scipy.stats for 80+ probability distributions, random sampling, and statistical tests

statsmodels for more statistical modelling, tests, regression, and data exploration

scikit-learn for machine learning: classification, regression, clustering

xlwings for controlling and interfacing with Excel

xlrd, **xlwt**, and **openpyxl** for reading and writing Excel spreadsheets

NetworkX - the standard Python library for network modelling and analysis.

Python tools for visualisation

Matplotlib is the workhorse Python 2d plotting engine

Seaborn is a higher-level package for statistical graphics that enhances Matplotlib

Altair is a Python interface to creating interactive charts using Vega-Lite and Vega. Uses a “grammar of graphics” approach.

Holoviews is the recommended high-level Python interface to creating interactive plots with Bokeh. It also supports Matplotlib and Plotly as plotting back-ends.

Bokeh is an upcoming browser-based alternative to Matplotlib. Low-level.

mpld3 is a simple extension to Matplotlib for rendering interactive plots in Jupyter notebooks using the D3 Javascript library (d3js.org).

VisPy is a visualisation library that makes use of GPUs for fast real-time rendering.

Chaco is a different graphics engine for interactive real-time plotting. Older but still powerful.

Python for high-performance computing

Numba is another JIT compiler for speeding up Python code (especially loops). Newer and less robust than Numexpr but super-easy to use and blazingly fast.

Numexpr is another (older) JIT compiler that can utilise multiple cores and avoid temporary array creation. Simple; not as powerful as Numba.

Dask for parallel task execution and working with out-of-core data sets.

IPython Parallel is a powerful, easy high-level interface for interactively running jobs on a cluster.

PyCUDA is a Python interface for performing fast array computation on NVIDIA's GPU devices

Celery is a task-queuing system for background processing and regular scheduled Python jobs

Python tools for big data and cloud computing

Dask: parallel task execution and working with out-of-core data sets.

boto library for interfacing with Amazon's web services from Python

MRJob: Python library for Map-Reduce (Hadoop) jobs

disco: similar in purpose to mrjob, but does not require Hadoop

Python for web development

Django: a large, opinionated web framework. Well-supported and well-documented.

Flask: lightweight “micro-framework” for web development in Python. Very easy to use, but still powerful and scalable. Flexible.

Zappa: a tool for automating “serverless” cloud deployments of Python code via AWS Lambda and AWS API Gateway

Commercial services

Here are some commercial services for revision control, CI, logging, etc. that can be useful for Python projects in production:

- **GitHub**, **GitLab**, and the Atlassian suite for code hosting and project management
- **Travis-CI** and other continuous integration tools
- **Sentry** for event logging
- **New Relic** for performance monitoring
- **Heroku**: simple Python integration for scalable

31.3 More exercises

In addition to the exercises and challenge exercises in these notes, you can try out the following resources:

- “Cracking the Code Interview: 189 Programming Questions and Solutions” (a book) by McDowell.
- CodingBat
- Hackathons

31.4 Questions?

You are welcome to contact us if you have any further questions after the course. You can reach us at info@pythoncharmers.com.

Further support

Python Charmers offers short-term and long-term consulting support with building data analysis tools, running simulations, optimising code, and providing customised training for specific analytical Python tools. Please feel free to contact us about your needs at support@pythoncharmers.com.

31.5 About Python Charmers

Python Charmers is the leading provider of Python training in the Asia-Pacific region, based in Australia and Singapore. Python Charmers specialises in teaching data scientists, engineers, scientists, traders, and computer scientists using the Python language.

Contact

- **Phone:** +61 1300 963 160
- **Email:** info@pythoncharmers.com
- **Web:** <http://pythoncharmers.com>

