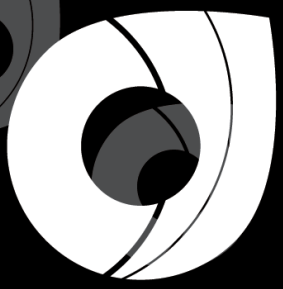
The Corndel DevOps Engineering Programme

in association with Softwire

Module 2: General Purpose Coding - Part II



Corndel Digital.

DA65

Introduction

One of the key tenets of the DevOps philosophy is that *software development is a team sport* and is best practiced by working collaboratively in cross-functional teams. In this module we'll cover **pair and mob programming** — two collaborative coding techniques born out of the Extreme Programming movement that embrace this philosophy and help to create better code as a result. Automation is another key pillar of DevOps and leveraging the power of **Application Programming Interfaces** (APIs) will allow you to extend automation to all aspects of the software development lifecycle. We'll discuss what APIs are, the various forms in which you'll encounter them, and how to make use of them so that you can interact with services programatically, opening the door to more advanced automation.

Lastly, we'll look at the principles of **clean code** and the practice of **refactoring**, which form part of the DevOps philosophy of continuous improvement and allow us to keep code quality high as the needs of a project evolve. The **object-oriented programming** (OOP) paradigm facilitates those goals, letting us design software architectures that are able to adapt to change and lead to readable, maintainable code. We'll therefore dive briefly into how OOP works in Python.

By the end of this module, you should be able to:

- Use pair and mob programming techniques effectively to improve collaboration and write better code.
- Understand what an API is, why they're useful, and how to interact with web services via REST and SOAP APIs to enable automation.
- Refactor your Python code to improve its structure and readability, making use of classes to provide suitable abstractions.
- Use trunk-based development to make parallel work easier to merge.

The reading material is covered in the sections listed below. As with other modules, the content is split into Core Material and Additional Material, which you can choose to undertake after completing the Core Material and Project Exercise. As always, we recommend that you try out the examples as you go along, since these will give you some valuable hands-on practice.

Table of Contents

Core Material

Additional Material

Chapter 1

Pair and Mob Programming

Coding is a team activity. There's sometimes a misconception that equates programming with typing; in reality, much more time is spent thinking about possible approaches and solutions, considering requirements, etc., than writing the code itself.

In the late 1990s, Kent Beck and colleagues introduced a new approach to software development, which they called *extreme programming*. One of the key principles of this methodology was the belief that programming is a fundamentally collaborative activity, and that the best code is often written when the task is shared, rather than done individually. The practice of **pair programming** (and, as an extension, mob programming) was therefore considered an important part of their approach, and has since been widely adopted.

Pair programming

Pair programming is essentially two people writing code together using one computer. In most scenarios, only one person controls the keyboard and mouse at a time, but both work together to implement a piece of functionality. Key aspects of pair programming are communication, trust and open-mindedness.

There are a number of benefits to pair programming:

- **Knowledge sharing:** having people work together on a piece of the code helps to spread technical and domain knowledge across the team on a daily basis and prevents silos of knowledge.
- **Reflection:** by having to explain and discuss our ideas out loud, we're encouraged to reflect if we really have the right understanding, or if we really have a good solution.
- **Maintaining focus:** by working closely with someone else, you are more likely to remain focused on the task at hand. That focus also helps to limit the amount of work in progress, which improves team productivity.
- **Continuous code review:** the code is constantly being reviewed by the navigator. Getting the review as you write the code means that any issues can be worked on straight away before they become a problem, and makes certain types of bugs much easier to spot.
- **Combined modes of thinking:** the driver is usually thinking "tactically" — how to write the current line of code, paying attention to the details — while the navigator thinks more "strategically", considering the bigger picture.
- **Collective ownership:** consistent pairing means that every line of code was touched or seen by at least two people, increasing the chances that everyone on the team will feel comfortable changing the code

almost anywhere in the codebase. It also makes the codebase more consistent than it would be with single coders only.

Taken together, these benefits can lead to better code quality and fewer bugs. However keep in mind that pair programming has its own pitfalls:

- It's easy for the person who is not coding to become disengaged with the task and play a passive role.
    - To mitigate this you can swap roles (see hints and tips below)
- When the paring task is straightforward it may be more productive to have people working on two separate tasks individually
- It may take considerable time to agree on how to tackle a given task

Pairing styles

While the general format of pair programming is the same, there are some variations that have a different focus on what they're trying to achieve. In all cases, the philosophy is the same: two heads are better than one, bringing together different perspectives and approaches that lead to better quality code as a result.

Driver and Navigator

These classic pair programming roles can be applied in some way or another to many of the approaches to pairing.

The **Driver** is the person sat at the keyboard typing in code. They are mostly concentrating on the minutiae of the code at hand, ignoring larger issues for the moment. A driver should always talk through what they are doing while they do it.

The **Navigator** is the person sat next to the driver. They review the code on-the-go, give directions and share thoughts. It sounds like they're on the sideline, but because the navigator isn't typing, they have more time to think and can therefore consider the bigger picture, and will often lead the problem-solving as a result. They should still be paying proper attention to the code the driver is writing and should be understanding and contributing to it.

Ping Pong

Closely related to Test Driven Development (TDD; something we'll cover in detail in the next module), "ping pong" pairing follows the principle of writing a test first, then writing the code to make it pass, with one person taking on each of those activities as the driver. A typical flow would be:

1. Person A writes a test
2. Person B writes the simplest code to make that test pass
3. Person B then writes another test
4. Person A writes code to make ***that*** test pass
   ... and so on.

Mentor and Student

While this may be the most familiar scenario in which you might have encountered pair programming, it should be stressed that it isn't the default style, and that pair programming is a much wider practice than simply training junior developers or new team members. That said, it can be an effective way to share knowledge, following the belief that it is better to learn by doing than by watching.

In this type of pairing, the more experienced develop will usually act as the navigator for longer periods of time, giving the driver more hands-on time at the keyboard. Swapping roles is still useful, but it's often for shorter periods, so that the mentor can show a specific technique or block of code, rather than the usual 50/50 split.

Pairing etiquette

- **Pay attention and be engaged.** Being engaged means no headphones and minimal phone checking. Pay attention to what your partner is doing. Ask questions and offer suggestions.
- **Encourage vulnerability and discourage judgement.** It takes courage to say "I don't know". Similarly, be empathetic toward your partner and create a space where they can be vulnerable too; judgements, put-downs, criticisms and so on have no place in pair programming.
- **Be humble and willing to try things.** A useful kind of humility in pair programming is "strong opinions, weakly held" — act on what you think you know, based on the information you have, but actively search for new information to deny or confirm your ideas and opinions.
- **Be a navigator, not a backseat driver.** Instantly calling out typos and asking questions like, "You're going to initialise that variable first, right?" can aggravate a driver and interrupt their thought process. It's better to give them a chance to find small things like this themselves and then offer suggestions only if they don't.
- **Recognise your emotions and be patient with your pair.** As humans, we have emotions and moods. And if we're not careful, those can spill over onto our partner. Have the self-awareness to recognise when these things happen, acknowledge them out loud, and focus on doing better. For instance, you can say, "Wow, the way I just said that was unkind. I didn't mean it to be. I'm just frustrated that this bug is taking so long to find." Conversely, remember that your partner is also a person with feelings. Grant them a little slack if their

emotions spill on to you. But as soon as it becomes a pattern, respectfully broach the subject with them.

Hints and tips

The following are some useful thoughts on pairing that we've gathered, primarily from our own experience of pair programming. Try to bear these in mind at all times when actively pairing.

- **Talk constantly.** That means both of you — make sure you're having a continuous dialog and are sharing all your thoughts with each other.
- **Question everything.** If you're not sure, ask — chances are you may be onto something!
- **Don't do anything without discussing it first.** You should both be agreed on the approach before writing the code.
- **Write down ideas for later.** When you think of something that needs more thought, but doesn't impact what you're working on right now, write it down on a post-it note. This allows you to maintain focus on the current task without forgetting important ideas.
- **Swap seats regularly.** It can be easy to lose track of time, so use a timer and swap over after a set period (30 minutes is a good length).
- **Teach each other keyboard shortcuts and coding tricks.** Don't sit there getting frustrated because someone is clicking through multiple menus or doing things naively — tell them how to do it faster and get each other into good habits.
- **Keep pairing tasks for pairing.** Don't split off and code on pairing tasks alone for a bit, because when you come back to the task the joint flow will have been disrupted and the other person won't ever get back up to the same level of involvement. If you're going to split off and work alone for a bit, work on something different.
- **Remember to take breaks.** If you're getting into the flow and talking all the while, pairing can be very tiring. Make sure you take regular breaks (this also allows both of you to do other, non-coding things, like checking emails, etc.).
- **Mix it up.** Pair with different people to spread knowledge, tips, tricks and experience around the team and the project.
- **Choose the hardest tasks for pairing.** Experience shows that the most benefit from pairing comes when attacking really hard problems, where the dialog between two minds cracks the problem fastest. Leave the more mundane tasks to solo programming,

because there's little benefit from two people working on them.

Mob programming

Mob programming sounds intimidating, but is really an extension of pair programming, where the whole team works on the same thing, at the same time, in the same space, and at the same computer. The difference from pair programming (apart from the number of people involved!) is that the activity covers more than just coding. Teams practising mob programming work together on almost all the work of a typical software development team, including defining stories, working with customers, designing, testing, and deploying software. The core idea is that the team focuses on one work item at a time, with everyone involved in its completion.

Mob programming can be useful when you want to involve multiple people (with differing interest/roles) in the development of a new feature:

- Any disagreements between sections of the team (design, user experience, testing) can be discussed and resolved early and in the open
    - This avoids the need for time consuming communication over email/messaging services
    - Issues that are raised can be addressed immediately rather than being painfully corrected later
- New features/concepts can be implemented as a group rather than by one party
    - This helps prevent people from objecting to large functional changes to the system as they have been an active part of designing these changes
    - The entire team has a great understanding of the new feature, allowing anyone to easily work on it in the future
- Responsibility for successes and failures are shared by the entire team
    - This makes it easier to make bold decisions such as architectural changes

Having so many people working focused on a single task allows you to effectively tackle big and difficult problems, but it is also a large investment of your team's resources, so make sure you pick tasks that are going to benefit from that level of effort. Keep in mind that mob programming also suffers from the same drawbacks as pair programming (if not more so).

- Larger groups are less likely to be able to agree
- Need to work to avoid the "loudest voices" swaying the team too much
- Keeping everyone in the group up to speed with what is happening can be very time consuming

Collaborative coding

These days, writing code as part of a team is the norm, and so collaboration, communication and sharing knowledge are critical skills. A team is at its most productive when knowledge — about both the technology and the business domain — is shared effectively among its members.

Most software teams are composed of developers with different levels of knowledge, including different domain expertise and programming experience. Team members can also come and go throughout the lifetime of a project, and new team members can struggle to absorb the information they need to ramp-up quickly. Senior team members often play a significant role in knowledge transfer, but typically have less time available. Finding effective ways to share knowledge can therefore make a big difference. This is especially true in cross-functional teams.

In addition to pair programming, there are a number of ways in which knowledge can be shared within your team, where all members feel they are able to contribute.

Code review

Code review is a common practice to check that code quality remains at a high standard on a project. The more senior developers on the team will typically review the code submitted by other team members and provide feedback before it is combined into the master project repository. In addition to ensuring code quality and addressing implementation issues, this is a great opportunity to share knowledge within the team, by pointing out existing parts of the codebase as an example to follow, or explaining best practices or implementation subtleties that are relevant to the submitted code.

Reading the code reviews of other developers on your team is also a good way to see what they're working on, how different people approach similar technical problems and whether there's a solution to the issue you are trying to solve already implemented somewhere else in the codebase.

Whiteboard sessions

Sometimes, you may have spent days working through a particularly thorny issue, or had to learn about a particular new technology or technique. This kind of knowledge is really useful to share with the rest of the team (someone else may have to deal with exactly the same issue in the future), but the idea of preparing a formal presentation can be off-putting. A much less intimidating option is to organise a *whiteboard session*. This is an informal meeting where people gather round a whiteboard and you share what you learnt from your experience, using the whiteboard to sketch out useful concepts and make impromptu notes. Try to make the session more interactive by encouraging questions or input from others. If you think the notes might

prove useful in the future, you can also capture them with a photo at the end of the session and save it somewhere the whole team can access.

Brown bag lunches are a similar format, where people bring their lunch (the brown bag is optional!) and eat together while discussing a particular topic, technical or otherwise. It allows team members to socialise and get to know each other in a relaxed environment, while promoting learning and building trust. Keeping it informal also makes people feel more comfortable asking questions.

Documentation

Most developers dread having to write documentation. This leads to documentation being patchy, poorly maintained, and often out-of-date. Instead of applying a rigid process for writing and reviewing documentation, it is often helpful to treat it as a living document that everyone on the team feels they are able to edit and improve. Taking a collaborative and iterative approach to documentation can lighten the burden and also helps to share that knowledge more effectively, since contributing to it is often the best way to become familiar with it.

Some forms of documentation can also be streamlined, or eliminated entirely, by adopting good coding and DevOps practices. Code should be self-documenting wherever possible, for example:

- Well-written code should be readable and not require a lot of additional explanation.
- Automated tests can serve as documentation of requirements.
- Data model diagrams can often be generated automatically from the code or database schema using available tools.
- Scripts and immutable infrastructure can remove the need for lengthy developer setup instructions.

Where documentation is required, READMEs within the project codebase are the best place for code-relevant information: keeping them in the same repo means they are more likely to be read and maintained.

Show & Tells

Rather than sharing knowledge within the team, these sessions provide an opportunity for the team to share their progress with the wider organisation. These sessions shouldn't be treated just as "progress updates", but a chance to talk openly about what techniques and approaches you've found to work well, and what challenges you've faced. You'll often find that other teams may be facing the same issues, or have found approaches that you can adopt; alternatively, they may find help from your team's experiences.

Chapter 2

Application Programming Interfaces

An Application Programming Interface (API) is an interface between multiple software systems. It provides a structured way to request that an action is taken (such as creating and deleting objects) or that data is sent. It defines the types of call one can make, how to make that call (parameters) and the effects of the call (e.g. receiving data in a certain format). By defining an API, a system allows for communication in an automated fashion, and the construction of strong pipelines.

The key benefits of automation are:

- Reproducibility
- Reducing human error
- Version control
- Saving time and money

The use of APIs plays a critical role in opening up new possibilities to automate processes and workflows. You've probably seen the term "API" come up before. Operating systems, web services and app updates often announce new APIs for developers. But what is an API, and why are they useful?

Application Programming Interface (API)

An application programming interface (API) is an interface or communication protocol between systems, whether they are web-based, operating systems, databases, software libraries or computer hardware. They are the means by which systems talk to each other, and define the ways in which you can interact with those applications.

APIs provide a means for different applications to interact with each other. Applications define their own APIs, providing a controlled way for their functionality to be exposed to the outside world. APIs define two roles: the **provider** and the **client** (or consumer). The provider is the application that, well, *provides* some functionality via its API, and the client calls the API in order to make use of that functionality.

One of the key benefits is that the consumer doesn't need to know *how* a given operation works under the hood — just *what* that operation achieves. Programming interfaces intentionally hide the implementation details so that users don't need to understand the complexities inside the system.

APIs are helpful because they:

- make life easier for developers

- control access to resources
- allow communication between services (and machines)

APIs come in many different forms and contexts:

- describe the expected behaviour (specification) that libraries or frameworks may implement
- specify the interface between an application and the operating system (e.g. POSIX, Windows API)
- allow remote systems or resources to be manipulated through defined protocols (e.g. SQL)
- enable interaction with other applications via the web (e.g. REST, SOAP)

In the context of this course, we'll mostly use the term "API" to refer to a specific kind of interface between a client and a web server, sometimes referred to as a *web API*. It's worth noting that this is only one type of API, though, and there are many others.

Web services and web APIs

The explosion in popularity of the World Wide Web in the late 1990s also lead to the emergence of services that leverage web technology, such as web servers and HTTP for communication. These web services have, in turn, come to dominate modern approaches to building applications and how they talk to each other. There are a few API standards that have emerged as the most widely adopted by web services, but before discussing them, it's worth talking a bit about the communication protocol that underpins most of them: HTTP.

Hypertext Transfer Protocol (HTTP)

You've probably heard of HTTP, but it's useful to know a bit about what it is and how it works, given that it plays such an important role in web APIs.

Fundamentally, the Hypertext Transfer Protocol is the messaging system that your browser uses to fetch web pages, images, etc., and to submit form data (for example, when filling in your details for a shopping order). Pretty much everything you do on the World Wide Web is sent using HTTP.

More formally, HTTP is an application layer transport protocol that forms the basis for almost all web traffic. It's a request-response protocol, meaning that a client sends a request to a server and the server sends a response back (as opposed to one-way forms of communication, such as sending an email). Your web browser is an HTTP client in this context. It sends a request to the web server containing the following details:

- Request method
- Uniform resource locator (URL)
- HTTP version
- Message containing

    - Headers
    - (Optional) body

The HTTP *method* indicates the desired action to be performed — whether to fetch something, or create it, or even delete it. For this reason, these request methods are sometimes referred to as HTTP *verbs*. The "something" in this context is called the *resource*. The most commonly used methods are:

- **GET:** get a resource from the server. If you make a GET request, the server looks for the data you requested (for example, a web page) and sends it back to you.
- **POST:** send data to the server - often causing the server to perform some kind of action (eg creating a new resource or triggering a side effect). The data is sent in the body of the request.
- **PUT:** send data to the server which replaces the corresponding data on the server (often used for updating or creating a resource). PUT requests are *idempotent* (meaning that making a call once is identical to making the call multiple times).
- **PATCH:** send data to the server which modifies part of an existing resource. A patch request is not required to be idempotent.
- **DELETE:** delete the specified resource on the server. Like PUT requests, DELETE requests are idempotent.
- **HEAD:** get details about a resource. A HEAD request is almost identical to a GET request, except the response doesn't contain the data itself, just details about that data. These requests are useful for checking what a GET request will return before actually making the GET request — like before downloading a large file.

When you visit a web page in your browser, the browser sends a number of HTTP GET requests to the web server that hosts that website, to fetch the page HTML itself, plus any images or other resources it needs. Conversely, when you fill in a form on a web page and click the "submit" button, your browser sends a HTTP POST request containing the data from that form. If the server accepts the POST request, it will create a new resource (for example, a shopping order).

The fact that POST requests are not idempotent also affects how web browsers behave. If you try to reload or go back once you've submitted a form, your browser will typically warn you that doing so will submit the form again — i.e., make the POST request again. This warning is because doing so may create another resource (in our example, a duplicate shopping order).

The resource in question in each of these requests is specified by a uniform resource locator – a URL. For a web page, that URL might be something like `http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol`, but URLs can also specify other things, like an account on GitHub (`https://api.github.com/users/CorndelWithSoftwire`), or some other resource. URLs are made up of several components. In the example of the above Wikipedia page:

- Scheme: **`http`**
- Host: **`en.wikipedia.org`**
- Path: **`/wiki/Hypertext_Transfer_Protocol`**

HTTP request messages omit the scheme (since it is implicit) and specify the host and path components separately. In the case of a GET request for this Wikipedia page, the request would look like this:

```
GET /wiki/Hypertext_Transfer_Protocol HTTP/1.1
Host: en.wikipedia.org
```

On the first line, the HTTP method is declared (`GET`), followed by the path to the resource from its URL (`/wiki/Hypertext_Transfer_Protocol`), and finally the HTTP version (`HTTP/1.1` means version 1.1 in this case). The following line declares the first (and in this case, only) request header. Headers are specified as `key: value` pairs, one per line. In this case the `Host` is specified as the host for the resource from its URL (`en.wikipedia.org`).

Let's look at a more complicated example, this time making a POST request with form data to create a new user:

```
POST /api/users HTTP/1.1
Host: www.example.com
Content-Type: application/x-www-form-urlencoded

firstName=Joe&lastName=Bloggs
```

Post vs Put

One common source of confusion is the difference between POST requests and PUT requests. Both of these methods involve sending data to the server in the body of the request, but they imply different things about what the server will do with that data.

A PUT request asks the server to take the data in the body of the request, and put it in the location specified in the URL. If there is already some data there, it will get erased and replaced (updating the resource) otherwise it will be added (creating the resource).

PUT requests are therefore idempotent - it doesn't matter if you make the request once, twice or multiple times, you will always get the same result - the resource you specified will contain the data you requested.

A POST request is much more general. The server can do whatever it likes when it receives a POST request. That might be to create or update a resource (similar to a PUT request) but it could also return some data and change nothing, or trigger some kind of side effect eg triggering a data export.

As a result, POST requests might not be idempotent - making the same call multiple times might cause multiple copies of the resource to be created.

In this case, the method is declared as `POST`, the resource path is `/api/users` and the protocol version is `HTTP/1.1`. In addition to the `Host` request header, the `Content-Type` header specifies the format for the data contained in the request — here, `application/x-www-form-urlencoded`, which is the default encoding used by browsers when sending form data. Finally, the *body* of the request appears after an empty line (the empty line is important!). Using the specified form encoding, the data in the body consists of name/value pairs for the `firstName` and `lastName` properties.

Now have a go at writing an HTTP request for yourself. We want to **update** the address for an existing user. The URL for that address resource is **`www.example.com/api/users/26/address`** and the properties to update are **`streetAddress`**, **`city`**, **`county`** and **`postcode`**. Think about which method is appropriate here and what the request body might look like.

HTTP response

When a web server sends a response, it contains the following:

- HTTP version
- Status code
- Message containing
    - Headers
    - Entity meta information
    - (Optional) body content

A typical HTTP response might look something like this:

```
HTTP/1.1 200 OK
Content-Type: text/html

<!DOCTYPE html>
<html>
...
```

On the first line, the protocol version being used by the server is declared (`HTTP/1.1`), followed by a numeric **status code** and its associated textual phrase (`200 OK`). The second line declares a response header, in this case indicating that the `Content-Type` of the response body is HTML. Finally, the response body starts on line 4 (the empty line before it is required, marking the end of the headers and the start of the body). Here, we've truncated the body, but it contains HTML (perhaps for a web page that was requested).

While web browsers don't usually display response status codes when the request is successful, chances are you've seen them when things go wrong -- either when the requested resource isn't found (404 NOT FOUND) or if the server is experiencing problems (500 INTERNAL SERVER ERROR). Actually, what you are probably seeing in those scenarios is the *HTML body* that is contained in the HTTP response, but the status code would match.

HTTP response status codes are 3-digit numbers indicating whether a specific HTTP request has been successfully completed. Responses are grouped into five *classes*, according to their first digit:

1. Informational responses (100–199): the request was received and the process is continuing.
2. Successful responses (200–299): the action was successfully received, understood, and accepted.
3. Redirects (300–399): the resource is located elsewhere, and further action must be taken in order to complete the request.
4. Client errors (400–499): the request contains incorrect syntax or cannot be fulfilled.
5. Server errors (500–599): the server failed to fulfil an apparently valid request.

A 404 status code, therefore, means that there was a problem with the client's request (in this case, they asked for a resource which doesn't exist), whereas a 500 code means a server error (perhaps due to a bug in the web server code). You can find a full list of HTTP status codes here.

Caching

Most websites and services have some notion of static data; i.e. data that changes very rarely. Common examples of static data includes

- Logos
- Images
- Styling (CSS)
- Javascript

Very often some of this data is shared between pages. This could mean that clients (i.e. browsers or consumers of web services)

Setting Status Codes in Flask

Flask methods (or view functions) can optionally return HTTP status codes as a second return value (recall tuples from the Module 1 workshop):

```
@app.route('/')
def homepage():
    return 'Hello There!', 200 # OK
```

This can be useful for returning error status codes:

```
@app.route('/book/<id>')
def get_book(id):
    try:
        return dictionary[id], 200
    except KeyError:
        return f"No book found with
id: {id}", 400 # Bad Request
```

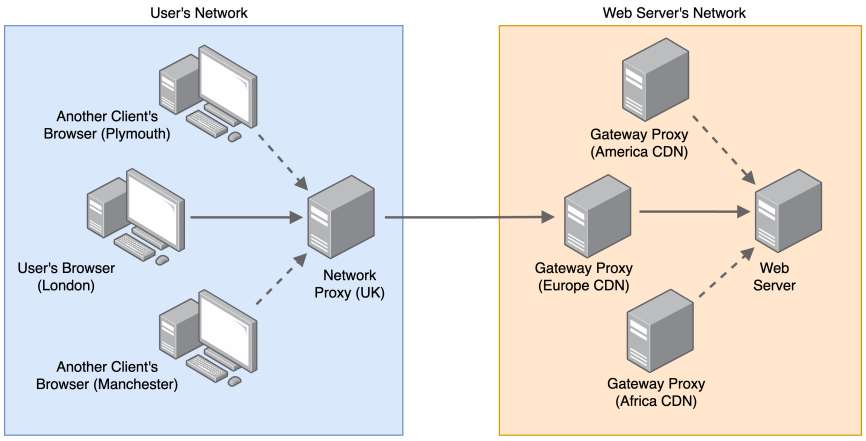
Flask provides some more detail on what can be returned from view functions here: https://flask.palletsprojects.com/en/1.1.x/quickstart/#about-responses

could end up unnecessarily downloading the same data multiple times.

This is problematic because:

- The client/server would end up using additional bandwidth
- Content loads slower than it needs to (the data has already been downloaded once, why not just reload it?)

Because of these inefficiencies the web provides several layers of ***caches*** which are used to determine whether to reuse data, or request a fresh copy upstream:



- The user's web browser (referred to as a ***private*** browser cache)
- The user's network proxy (if present), one type of a ***public*** shared cache
- The gateway proxy or Content delivery network (CDN), another type of a ***public*** shared cache

This isn't an exhaustive list, however.

The behaviour of these various caching layers can be managed through the HTTP `Cache-Control` header. This can be specified

Content Delivery Networks (CDNs)

When a user visits our site, their browser communicates with our server by physically sending signals back and forth to load the web page. Those signals travel fast but they aren't instant. The further away the user gets from our server, the more 'latency' (the time between them requesting something from our server and getting a response) they will see. If the user is a long way away, this latency can start to become an issue. The role of a Content Delivery Network (or CDN) is to get content to users as quickly and reliably as possible. They typically do this by having:

- a network of servers that is spread out all over the world, so that a server is always near the user, meaning latency is low
- infrastructure to ensure that they can always provide the content (high availability)
- infrastructure to ensure that they can provide that content to a many users simultaneously

If our site contains content that doesn't change often (such as images, videos, CSS files etc) then we can send that content to a CDN. Every time a user wants that content, they can collect it quickly and easily from the CDN instead of needing to go all the way to our server. We'll still need communication with our server to handle any logic or to perform any actions, but being able to use the CDN for static content improves experience for our users and reduces the load that is placed on our own server. As a result, CDNs now serve a large portion of the content on the internet.

on both request & responses for data but are more commonly used on responses

- This header provides several directives including:
    - Declaring whether data can be cached at all: `Cache-Control: no-store`
    - Which caches can cache the data: `Cache-Control: public`
    - When the cached data becomes stale and needs to re requested: `Cache-Control: max-age=6000`
    - How to validate stale cached data one it has expired: `Cache-Control: must-revalidate`

Validation Headers

Re-validation occurs when either cached data becomes stale or the user explicitly refreshes a page.

To indicate how recent response data actually is, caches usually provide one or two of the following validation headers

- `Last-Modified` (when the data was last modified)
- `ETag` (a version number for the cached data)

When re-validating data, the client can then ask if there has been a change since `Last-Modified` (using the `If-Modified-Since` request header) or whether there is a newer version than the `ETag` (using the `If-None-Match` request header)

- If there isn't a new version to return then the response will have a HTTP status code of 304 (unmodified)

For more information and advice on caches please checkout this link: https://www.mnot.net/cache_docs/

Cookies

An important property of HTTP is that it is ***stateless***. This means that each request is fully self-contained and does not depend on previous requests, so requests could be sent in any order and still be valid. This makes the server's life much easier, as it doesn't need to remember request details for the future. It also means that the web server application can run across multiple machines, and that each one can respond to requests from any client, since each request is self-contained.

That's the theory. In reality, some state often *does* need to be maintained between requests. A prime example is authentication, since you don't want to have to log back in for every single page that you request from a website. This is where **cookies** come in.

A cookie is a piece of information returned in an HTTP response that the client stores and then includes in every future request to the same server. Crucially, the cookie is stored and maintained by the client, not the server. Clients associate cookies with servers based

on their domain (host) name. A typical sequence of requests and responses using cookies might look like this:

1. The first time a client sends a request to a particular server, there are no cookies.
2. When the server responds, it includes a `Set-Cookie` header that defines a cookie.
3. The client saves this cookie (the storage for cookies is often called a "cookie jar").
4. When the client makes another request to the same server, it includes a `Cookie` header containing the same cookie data.
5. The server can use that cookie data to identify the client or retrieve other client-related info.
6. When the server responds, it won't include the `Set-Cookie` header again unless it wants to change the cookie data.

What info does a cookie contain?

- Name and data
    - The data stored has a maximum size limit imposed by the client (typically 4 KB or less).
    - A server can set multiple cookies with different names, but clients usually limit the number of cookies per server (around 50).
- Domain. A cookie is only included in requests matching its domain.
- (optionally) Expiration date. Browsers can delete old cookies.
- (optionally) Additional properties such as Path, HttpOnly, Secure and SameSite that affect how and when the

The SameSite Cookie Property

SameSite is a property on cookies supported by most modern browsers. Originally cookies would be sent by all requests to the applicable domain. This can lead to a potential security exploit called Cross Site Request Forgery (CSRF) where a hacker's site can make a request to, say, a bank's site, sending over the user's authentication cookie for the bank. CSRF is covered in more detail in the data and security modules later in the course.

The SameSite property can be set to the following values:

- **None** (i.e. the old default mentioned above)
- **Lax** (the new default)
    - This means it will only send the cookie if the browser's search bar URL (technically the top level navigation) equals the cookie's domain.
- **Strict**
    - This means it will only send the cookie if the browser's search bar URL (top level navigation) equals the cookie's domain AND it hasn't been navigated to from a different domain (e.g. by a link from another site)

You can read more about SameSite cookie support here: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/

cookie will be used (we'll talk more about these in the security section of the course)

Cookies are used for many things, some very useful — like storing an authentication token, so you don't need to log in for each request, or storing your shopping basket items — and some potentially annoying — like assigning a tracking identifier so that advertisers can follow your browsing history across sites.

The server decides what content it wants to store in the cookie(s), while the client decides whether it wants to send those cookies in subsequent requests (by default it will, but things like private browsing mode will prevent previous cookies from being sent, which is why a website won't remember you when you start a private browsing session).

Many cookies (especially authentication tokens) have tamper prevention measures in place to avoid issues like broken access control. Examples of such measures include:

- **Encrypting the cookie**: This is where the server encrypts/decrypts the cookies being sent/received from users.
    - This completely prevents the user from seeing the original contents of the cookie (which could potentially expose attack vectors).
    - Furthermore this prevents users from easily modifying the cookie without rendering it invalid.
- **Signing the cookie with a hash**: This is where a code is generated from the original cookie on the server using a function called a ***hashing function***.
    - When the cookie is sent back to the server it checks that the cookie matches its hash by reapplying this function.
    - This prevents the user from modifying the cookie as, if they did modify the cookie, the code would change and the server would reject the cookie.
    - A common example of this is producing a Message Authentication Code (MAC).

You can see what cookies a server has set in your browser by going to your browser settings. Instructions for where to find them in several popular browsers can be found here, but all browsers should provide a similar way to view them.

Securing connections using HTTPS

HTTP is not encrypted, so is vulnerable to man-in-the-middle and eavesdropping attacks, potentially allowing access to sensitive information or modifying content (such as injecting malware or adverts into web pages). This security weakness lead to the introduction of the catchily named **Hypertext Transfer Protocol Secure**

(HTTPS), which is designed to withstand these types of attacks.

When your web browser makes a request for a URL that has the HTTPS scheme (i.e. starting with `https://` instead of `http://`), it signals the browser to use an added encryption layer, called *Transport Layer Security* (TLS), to protect the traffic. The HTTP request and response content remain largely the same (save for some additional headers), but they pass through this encryption layer before being sent. For this reason, the HTTPS protocol is sometimes called "HTTP over TLS".

Note: HTTPS connections can also be established using an older form of encryption layer called *Secure Sockets Layer* (SSL), which is essentially an older version of TLS and has largely been replaced by TLS, which fixes some vulnerabilities discovered in SSL.

The use of HTTPS provides a few key benefits:

- Authentication: the client can verify the identity of the server it is communicating with (and also vice-versa, though this is less common).
- Privacy: the data exchanged while in transit cannot be read by an eavesdropper.
- Integrity: the exchanged data cannot be modified by an eavesdropper while in transit.

HTTPS has seen significant adoption over the past few years and is now used more often by web users than the original non-secure HTTP, primarily to protect page authenticity on all types of websites, to secure accounts and to keep user communications, identity, and web browsing private. Whenever you are using a web API, you should check that you are calling it over HTTPS. If a service doesn't support HTTPS connections to its API, you should certainly think carefully about whether you want to use it — and what the security risks might be if you do.

Exercise: hand-rolled HTTP requests

If you'd like to experiment with sending out HTTP requests by hand there are a few dedicated tools that can you can use:

- Postman (see "Exercise: Dog API" below for an exercise of using this with an API)
- Fiddler

In reality, modern programming languages have plenty of libraries to make HTTP requests, so you'd never construct the raw requests yourself, but it's useful to see how they work under the hood and that there isn't anything particularly mystical or mysterious about them!

Common API standards and protocols

Types of web API

Web APIs typically fall into one of two categories: *Simple Object Access Protocol* (SOAP) or *Representational State Transfer* (REST). We'll cover both briefly, but focus more on REST, since you're more likely to encounter it when using newer web services.

Simple Object Access Protocol (SOAP)

SOAP is a standards-based protocol for access to web services that's been around for ages (in Internet years). It was originally developed by Microsoft in 1998 and isn't as simple as its name would suggest. The key concept in SOAP is that web apps provide one or more *services* that support a number of *operations*, which can be thought of as method or function calls. These operations are called by sending a SOAP message

SOAP relies entirely on XML for the messages that pass to and from the web service you're talking to. It was designed to be highly extensible, but you only use the pieces you need for a particular task. For example, when using a public web service that's freely available to everyone, you don't really have much need for the WS-Security extension.

A SOAP message consists of the following parts:

1. Envelope (required): the starting and ending tags of the message.
2. Header (optional): contains the optional attributes of the message. It allows you to extend a SOAP message in a modular way.
3. Body (required): contains the data that the caller sends to the server, or that the server transmits to the receiver.
4. Fault (optional): carries information about any errors that occurred during processing of the message.

Here's an example taken from the W3C SOAP documentation:

```
<env:Envelope xmlns:env="http://
www.w3.org/2003/05/soap-envelope">
  <env:Header>
    <n:alertcontrol xmlns:n="http://
example.org/alertcontrol">
      <n:priority>1</n:priority>

<n:expires>2001-06-22T14:00:00-05:00
</n:expires>
    </n:alertcontrol>
  </env:Header>
  <env:Body>
    <m:alert xmlns:m="http://
example.org/alert">
      <m:msg>Pick up Mary at school
at 2pm</m:msg>
    </m:alert>
  </env:Body>
</env:Envelope>
```

As anyone who's used XML a lot will tell you, it can be hard to work with; it's hard to read and easy to get wrong. In the case of SOAP, the XML used to construct messages can

become extremely complex, which is problematic because SOAP is also intolerant of errors. Fortunately, many programming languages have libraries that will do the heavy-lifting of constructing the XML for you.

A key feature of SOAP is its use of Web Services Description Language (WSDL) documents to describe the functionality offered by the API. WSDL documents are also written in XML and provide a machine-readable description of how the web service can be called, what parameters it expects, and what data structures it returns. They use the XML Schema language to provide a precise definition of the data contained in the input and output message bodies, including the data types, which allow the caller to validate the request message structure before sending it and to know the precise structure of the messages it can expect to receive in response.

One of the most important features of SOAP is its built-in error handling. If there's a problem with your request, the error response contains information that you can use to fix the problem. Given that you likely don't control the web service you're talking to, this feature is particularly important, otherwise you would be left guessing as to why things didn't work. The error reporting even provides standardized codes so that it's possible to automate some error handling tasks in your code.

An interesting SOAP feature is that you don't necessarily have to use it with the HTTP protocol. There's even a specification for using SOAP over Simple Mail Transfer Protocol (SMTP) and there's no reason you couldn't use it over other transport protocols, too.

Representational State Transfer (REST)

REST is not a protocol, but an architectural style or design pattern for APIs. We call APIs that conform to this pattern *RESTful*. A RESTful web application exposes information about itself in the form of information about its *resources*. It also enables the client to take actions on those resources, such as create new resources (i.e. create a new user) or change existing resources (i.e. edit a post).

The name REST comes from the concept that the server will *transfer* to the client a *representation* of the *state* of the resource being requested. Client requests are defined by two bits of information:

1. The **resource** you are interested in. This is specified by the URL for that resource, also known as the **endpoint**.
2. The **operation** you want the service to perform on that resource. This is specified in the form of an HTTP method, or **verb**.

For example, let's say you wanted to fetch a specific Twitter user's details using Twitter's REST API. In this case, you'd make an HTTP request specifying the URL (uniform resource

locator) that identifies that user on Twitter and with the HTTP method GET.

A few of the core concepts of RESTful APIs are:

- **Stateless:** the server doesn't remember anything about the client who uses the API. It doesn't remember if the client already sent a GET request for the same resource in the past, it doesn't remember which resources the client requested before, and so on.
    - This means that each request must contain all the information the server needs to perform the action and return a response, regardless of previous requests made by the same client.
- **Uniform interface:** requests from different clients should look the same; for example, the same resource shouldn't have more than one URI.
- **Cacheable:** responses should contain information about whether the data they send is cacheable or not. Cacheable resources should be indicated with a version number so that the client can avoid requesting the same data more than once.
- **Layered system:** the request and response should remain the same regardless of whether there are several layers of servers between the client and the server.

Another key goal of RESTful APIs is to convey *semantic meaning* by using the properties of the underlying HTTP protocol itself. The action (or operation) performed by a given endpoint is conveyed by using the appropriate HTTP method, i.e.

Action	**HTTP method**
Create	POST (or sometimes PUT)
Read	GET
Update	PUT (or sometimes PATCH)
Delete	DELETE

Hypermedia as the engine of application state (HATEOAS)

This sounds very cryptic, so let's break it down: by application we mean the web application that the server is running. By hypermedia we refer to the hyperlinks, or simply links, that the server can include in the response. The whole sentence means that the server can inform the client , in a response, of the ways to change the state of the web application. If the client asked for a specific user, the server can provide not only the state of that user but also information about how to change the state of the user, for example how to update the user's name or

how to delete the user. It is easy to think about the way it's done by thinking about a server returning a response in HTML format to a browser (which is the client). The HTML will include tags with links (this is the hypermedia part) to another web page where the user can be updated (for example a link to a 'profile settings' page). To put all of this in perspective, most web pages do implement hypermedia as the engine of application state, but the most common web APIs do not adhere to this constraint. To further understand this concept, we recommend watching this 30 minutes YouTube video.

Comparing REST to SOAP

It's pretty much impossible to have a RESTful SOAP API:

- This is because HTTP SOAP services normally use POST for requests (technically speaking GET is possible but is seldom used) and do not support other HTTP methods.

As a result, even though REST isn't a protocol, direct comparisons are made between REST & SOAP.

Generally speaking, more modern web services tend to provide RESTful APIs, since they offer several benefits over SOAP APIs for most common use cases:

- REST provides the flexibility of different data formats (such as JSON) while SOAP is really intended for XML only.
- SOAP bubbles end up bloating the size of requests/responses.
    - Furthermore the standard of prefixing xml elements with namespaces tends to bloat up the content of messages.
- SOAP tends to have a higher barrier to entry that REST (with a REST API you sometimes make GET requests to unsecured endpoints in your browser)
- REST takes advantage of the unique aspects of the HTTP protocol by using a range of methods (GET/POST/DELETE) while SOAP is HTTP agnostic.

Even so, SOAP is still used by default in Microsoft's Web Communication Framework (WCF) for building web services (but it's easy to switch over to REST with a couple of config changes).

- In fact some services provide both REST & SOAP APIs to let consumers choose how they would like to communicate.

Exercise: Dog API

Everyone knows that the Web is powered by pictures of adorable animals, so we'll use the Dog API — "the Internet's biggest collection of open source dog pictures" — to try out some basic REST API concepts.

Let's start by getting a list of all available breeds. According to the documentation, the relevant API endpoint is `https://dog.ceo/api/breeds/list` — use Postman to make a GET request and see what response you get.

You'll see that the response data contains a **`message`** value, listing all the available dog breeds, and a **`status`** value, indicating if the request was successful or not.

Now try listing all sub-breeds for a given breed. What happens if you try calling the endpoint for a breed that doesn't exist?

Finally, try getting some cute dog pictures — try calling the random image endpoints for a specific breed or sub-breed. The response should contain a URL for a randomly selected image. Clicking that image URL in Postman will create a new GET request with that URL, and sending that new request should fetch you the image!

A couple of things to note:

- The API uses *nested* resources, a common API design pattern where sub-collections (in this case, sub-breeds) of a given resource (breeds) are located at a sub-path relative to the parent resource. For example, `/api/breed/terrier/yorkshire/images` where Yorkshire Terrier is a sub-breed of Terrier. This design helps to convey hierarchical relationships (like sub-directories in a file system).
- The `/images` endpoints are an example of API responses that contain URLs as a means of directing the client to fetch other resources (in this case, the image itself).

Using Postman

Postman (https://www.postman.com) is a great tool for exploring web APIs without having to write any code. We'll be using Postman to test out some of the APIs we'll be calling, so it's a good idea to spend some time becoming familiar with how to use it.

You can download it here (https://www.postman.com/downloads/) and learn how to make a basic request by following their Getting Started guide.

GraphQL

A GraphQL API also communicates over HTTP like REST, but unlike REST every request is done over POST and the type of request is specified in the body of the POST request.

- The response is regular JSON however the request body JSON only contains one field ("query"), which consists of the request written in the GraphQL query language

GraphQL attempts to deal with the problem of ***overfetching*** data by the allowing the client to specify exactly what it wants to consume. This is probably best explained by an example. Suppose you wanted to lookup a books's page count using a library's API:

- In REST, they might offer a GET endpoint that returns data about a given book (name, IBAN, publisher, etc.) along with the page count. Since this is the only endpoint that returns the page count for a book, you're forced to use this endpoint and throw away the data you didn't care about (e.g. the publisher field).
- In GraphQL, however, you could state that you're only interested in the page count and the server would be required to only return this field:

```
Request: { book(id: "1000") { pageCount } },    Response: {"data": {"book": {"pageCount": 1024 } } }
```

This is useful if you want to:

- Reduce IO between the server and client.
- Reduce server/DB CPU load by only looking up data when explicitly asked to.

GraphQL also has features to prevent ***underfetching*** where there is not a single endpoint that provides what the client needs (see this link for an example)

While it's often advantageous for the client to have a GraphQL API available, this often places additional burden on the developer of the server to be more flexible than a traditional REST API.

GraphQL is a relatively recent query language but thankfully client/server libraries are available in many popular languages: https://graphql.org/code/

A good tool for testing GraphQL APIs is the Insomnia client (think of it as GraphQL's equivalent of Postman).

Data formats

For web APIs, the most common data formats for request and response are XML and JSON --- though many other formats are used (anything from HTML, CSV, to plain text or binary).

If you're unfamiliar with XML or JSON, it's worth reading through these brief overviews here and here.

Indeed, many services with REST APIs will support multiple media types (data formats); it's up to the client to choose which they'd prefer to use, via a process called *content negotiation*. In this case, the media type is specified in the HTTP request header using the **Accept** and **Content-Type** attributes. The Accept header specifies the format of response data that the client expects and the Content-Type header specifies the format of the data supplied in the request body.

Authentication

Most web APIs you will encounter typically require some form of authentication, even if the service they provide is free to use. The primary reason for this is to prevent malicious usage, e.g., distributed denial of service (DDoS) attacks, which can be mitigated by revoking the authentication credentials for that user account. More generally, authentication allows a service to restrict what resources you are able to access (authorisation) and track activity for auditing purposes.

There are a number of ways in which API requests can be authenticated. Here's an overview of some of the most common:

Basic HTTP authentication

- Username and password are encoded (not encrypted) and sent in an HTTP request header
- The request header takes the form `Authorization: Basic <credentials>`, where `<credentials>` is the base64 encoding of the username and password joined by a single colon (`:`).
- One of the oldest forms of authentication.
- Generally considered a weak form of authentication, because:
  - The username and password are sent unencrypted (base64 is a reversible encoding), so the request *must* be sent over HTTPS, otherwise the credentials are at risk of being stolen.
  - The credentials have no expiry (apart from changing the password), so are vulnerable to replay attacks.
  - Credentials are usually cached by web browsers and there is no way for the server to tell the client to "log out" the user.

API key or secret

- A hard-coded token that is generated by the service for your account
- The key may be sent as a query parameter or in a request header, depending on the API
- Key has no expiry (unless you regenerate it)
  - Simpler to use

- More open to compromise as users will need to grant full access to applications that wish to use the API on the users behalf

OAuth

- Anyone using social sign-in (Google, Facebook) will be familiar with this
- When we say "OAuth", we almost always means OAuth 2.0 these days
  - OAuth 2.0 attempts to address some of the complexities of the first version of the protocol such as
    - Restricting the protocol to SSL/TLS (HTTPS)
    - Less back and forth communication between the application and the authentication server (compare the OAuth2.0 diagram here with this)
- Token-based auth -- authenticate using a separate identity service (e.g. Facebook) which returns an *access token**
- Token is passed in auth header on API requests
- Service verifies token (possibly by calling identity service to validate)
- Tokens are quite short-lived, but OAuth allows applications the ability to refresh them using refresh tokens
- Facilitates Single Sign On (SSO) by using a common authentication service across several web applications

OpenID Connect (OIDC)

- Similar in principle to OAuth
- Adds concept of *identity* as well as authentication
- Identity service returns both access token and *identity token*, which can contain details about the user, or be exchanged for full profile details by calling the identity service with that ID token

IP whitelisting

- Different approach to security than account-based authentication
- Only requests from a whitelisted set of IP addresses (or IP ranges) are accepted -- all others are rejected
- Hard to spoof (unless able to access network via, e.g., a Virtual Private Network)
- Access is more restricted
- Cannot call API from anywhere, so not well-suited to personal use (cannot call from other locations/networks)
- Does not identify individuals (unless your IP uniquely identifies you)

A word about tokens

Some auth services return a random string (hexadecimal or base64) as the token to be used. In this case, the service accepting that token needs to call the auth service to validate it.

More recently, *JSON Web Tokens* (JWTs) have become the standard.

- These are encoded JSON blobs containing more structured data and cryptographically signed by auth service.
- Other services are able to confirm validity of these tokens without needing to call auth service each time, by checking signature.
- When developing applications that use third-party OAuth or OIDC provider, debugging issues can be simplified if that provider uses JWTs.
- Can manually inspect the payload of a JWT using several online services, such as jwt.io

JSON web tokens (JWTs)

A JSON web token (JWT, often pronounced "jot") is a standard format that allows for self-contained authentication tokens.
They contain 3 parts:

- **header**: metadata about the token - such as what signing algorithm it uses
- **payload**: the content of the token - such as the name of the user, expiry date for the token, what permissions the user should have etc,
- **signature**: a 'signed' version of the payload that allows verification that the payload is valid

JWTs are often used to enable sign in across multiple systems. We can have a single Auth service that generates these tokens for our users, making the signature by encrypting the payload using a key that it keeps secret and nobody else knows (a private key). It also publishes a second key - a public key. The public key is useless for encrypting the payload (and so can't be used to make new tokens) but it can be used to decrypt the signatures signed by our private key. If the decrypted signature matches the payload, then we can be confident that the token must have been generated by the Auth Service and that the information in the payload hasn't be tampered with.

Other services can therefore use the public key to validate tokens without even needing to communicate with the Auth service. This gives us a number of advantages:

- other services can validate tokens themselves without needing to communicate with the Auth service - this improves performances and reduces load on the Auth Service
- the token itself can store any data we need - further reducing the need for communication between services
- its stateless - meaning we don't need to store information about which users are currently signed in / what sessions are active.

As always, there are also some downsides!

- its hard to revoke access to the site once you've given someone a token (typically you have to wait until that token expires)
- the size of the token is large compared with other methods

If you want to find out more, try experimenting with JWTs using services such as jwt.io.

Exercise: Practice with a dummy REST API

Use the 2.2 – Application Programming Interfaces#Postman tool to access the following API: https://rem-rest-api.herokuapp.com/:

- Experiment with the following HTTP methods: GET/POST/PUT/DELETE
- Next attempt a GET request with query parameters
- *(Stretch) can you integrate with this API in python?*

Discovering and understanding APIs

A challenge when using a third-party service's API is that it might not be obvious that it exists in the first place. Discovering the availability of APIs can be tricky, and once you know they exist, finding out how to call them/details of request and response can also be hard.

Tips for finding APIs

- **Google!** A quick search for "service name API" will often turn up details of its existence (and probably documentation, too).
- **Auto-generated documentation.** Services increasingly employ some form of auto-generated documentation for their APIs, which saves time having to manually write and update documentation. The most well-known example is **Swagger**, based on the OpenAPI specification.
- **REST API discovery (HATEOAS).** One of the core principles of REST is that of discoverability, and APIs that are truly RESTful will provide details of how to use the API in the response data itself, including information about other operations/resources that can be called. It's worth noting, though, that many services providing REST APIs do not implement this aspect (so are not "fully RESTful"). You can read more about the specification here.
- **GitHub or other open source repositories.** Even if not well-documented, the source code for a project itself provides unambiguous (if harder to understand) definition of the API it provides

Example of API Documentation

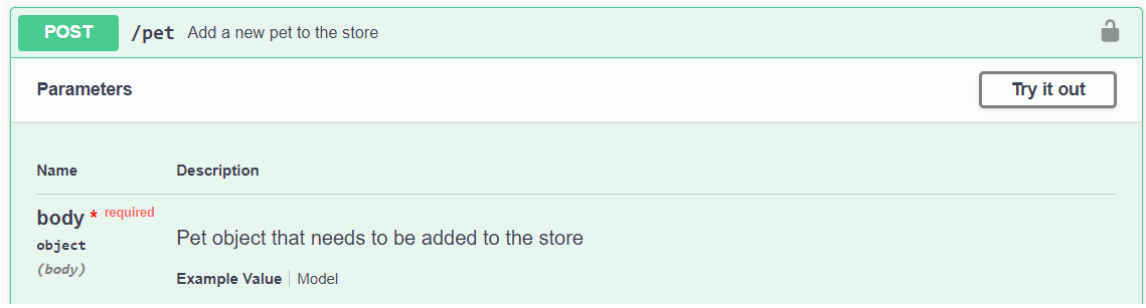
You can find an example of interactive swagger documentation here: https://petstore.swagger.io/

Note that actions are grouped by resource first and then by url path (colour coded by HTTP method).

Clicking on a url path then provides a summary of how calling it works:

- For GET requests swagger provides a list of parameters and descriptions for each one
- For POST requests swagger either provides an example of the post body (or a list of form parameters for form data submission)

For all paths, the possible response status codes are listed. Any deprecated paths are greyed out. There is a "Try it out" button you can click, which will send a request over to the API for you to try (see below):



You can edit your request before sending it if you'd like to.

After clicking the "Execute" button swagger will then make the request to the API and then present the results. In this way, swagger is acting very much like postman! At the bottom of the page there is a list of data models used in the API. These can also be accessed when clicking "model" on the paths that use them (see right).

Using APIs: best practices

Frameworks/libraries

Most modern languages have package managers to handle downloading/installing libraries and frameworks (in the case of Python, this is `pip`). Those tools can be used to automatically install/update packages as part of a build process. Once a package is installed, its API can then be called by your code.

It's important to be aware of the importance of versioning when using packages. Not all versions of a package behave the same (breaking changes are introduced in updates, bugs appear or get fixed), so if everyone on your team is using a different version --- or worse, your live application uses a different version than you're using locally for development and testing --- chances are that sooner or later you'll run into an inconsistency in its behaviour that can introduce bugs in your application. In the worst case, those bugs could even be security-critical! For this reason, most package managers also provide a way to document the version of each package that is being used by your code, and you should definitely add this file to your version control system (e.g. Git repo) so that it forms part of your project and so that any changes to it are tracked (for example if you update a package to a newer version).

Another important practice is to make sure you are updating packages when appropriate. Studies show that many projects adopt a "set it and forget it" attitude to package versions, meaning that once a given version of a package is added as a dependency to a codebase, it is rarely, if ever, updated to a newer version in the future. While there may be good reasons not to update (e.g. a breaking change is introduced to a feature that you are using), security is a very compelling reason to update. There have been many high-profile examples of major security flaws being discovered in some widely-used libraries or frameworks that have made the software that use them vulnerable to attack.

It's generally good practice to update package versions regularly because:

- it may fix existing bugs or provide new features
- changes to API usage can be accounted for gradually, rather than needing a major re-write in the future
- using a more recent version makes it trivial to rapidly update to latest version in the event that a major security issue is uncovered

However, some things to be aware of:

- new versions can introduce bugs too!
- testing may need to explicitly cover areas where updated package may impact (though good automated testing should minimise this!)

- if breaking changes are introduced, you'll need to update your code to reflect that

Web APIs

Many popular services providing web APIs now also provide software frameworks or libraries that handle the HTTP calls to those APIs for you, wrapping them in a software API that you can call directly from your code. This simplifies their usage significantly, so it's always worth checking to see if a library — often referred to as an *API client library* — already exists.

Keeping secrets

For APIs requiring authentication, it's important to think about *where* you're going to store the credentials. Hard-coding them in the code itself might seem like a reasonable option at first, but there are several reasons why this is considered a **Very Bad Idea**!

- Anyone who has access to the code can read those secret credentials. And for practical reasons, access to code repos often isn't as restricted as to other data stores (for example, all developers on the team can access them, whereas you might only want certain people to have access to the credentials used by your production app).
- Those credentials are harder to change. If they get stolen or invalidated, you have to make a code change and then release that change in order to update the production app.

For these reasons, secrets, passwords and other credentials shouldn't be stored in the code itself, but form part of the *configuration*. There are several places where that configuration might be stored:

- Environment variables (we'll discuss these more in module 4)
- Remote secret store or vault
- Manually maintained configuration files that are excluded from version control (in the case of Git, they should be listed in the `.gitignore` file to prevent them being committed by mistake)

There are pros and cons to all of these options, but we'll see in later modules why the first two options are generally preferable.

Failing gracefully

While this is an approach that you should adopt for all error handling, it's especially true when interacting with a web API, where the chance of errors can be a lot higher due to network issues or the service itself being down. You should think carefully about how to handle request timeouts and other error responses when calling a web API: some errors may indicate a temporary problem, while others (such as `4xx` HTTP responses) indicate a problem with the request itself, so trying again probably won't help.

Some general practices to consider are:

- Retry logic: repeat the request after some delay, typically with a maximum number of retries.
- Request throttling: limit the rate of requests to prevent overwhelming the external service (though many APIs will defensively apply rate-limiting themselves to mitigate attack, anyway).
- Circuit breakers: this is a design pattern to "throw a circuit breaker switch" (i.e. stop making calls to the service) if previous calls have failed above some threshold. Details of the pattern can be found here & here.

These techniques are common enough that there's a healthy ecosystem of libraries that implement them for you. In fact, API client libraries may already do this. As a general rule, look for an existing library or framework that might do the job for you, before trying to write your own.

Chapter 3

Object Oriented Programming in Python

Classes and objects

The concept of **object-oriented programming** (OOP) is a fundamental programming paradigm and underpins many key libraries, including the Python standard library itself.

Objects are a way of grouping together the properties and functionality of a conceptual entity. Many times they are based on objects in the real world. A classic example is that of a bicycle. In OOP, a bicycle object may have attributes, such as the make and model of the bicycle, the number of gears it has, and the currently selected gear. That bicycle object might also have functions (referred to as **methods**) that can change the currently selected gear, speed up, or apply the brakes.

Other concrete examples: an object could represent a person with a name property, age, address, etc., and behaviours like walking, talking, breathing and running; or an email object could have properties like recipient list, subject and body, and behaviours like adding attachments and sending. Objects can also model the *relationships* between things, such as companies and employees, or students and teachers.

Classes can be thought of as the *blueprints* for creating objects (an object is sometimes referred to as an **instance** of a class).

Abstraction and Encapsulation

These are two of the core concepts in object-oriented programming. Despite their cryptic names, the concepts are relatively simple.

Abstraction means using simple things to represent complexity. We all know how to turn the TV on, but we don't need to know how it works in order to enjoy it. In OOP, abstraction implies using simple things like objects, classes and variables to represent more complex underlying code and data. This is important because it allows code to make use of an object without having to know the details of how it is written internally (how it is *implemented*), which means that you can build up increasing layers of complexity by wrapping each conceptual piece of functionality as its own object (so objects that make use internally of other objects, and so on). As an example, you could have a `car` object that internally makes use of an `engine` object, which in turn has code that calls methods on various component objects ("battery", "spark plug", etc.).

Encapsulation describes the idea of bundling together data and methods that work on that data within one unit — i.e., a class. This means that all the code relating to that data is contained in one place. This concept is also often used to hide the internal representation, or state, of an object from the outside world (known as

information hiding). The general idea of this mechanism is simple. If you have an attribute that is not visible from the outside of an object, and bundle it with methods that provide read or write access to it, then you can hide specific information and control access to the internal state of the object. A car object limits your control of the engine to the ignition key and accelerator pedal — it doesn't let you tweak the fuel mix or valve timings.

Defining a class

Classes are defined in Python using the **`class`** keyword. Let's look at an example definition for a Bicycle class.

```
class Bicycle:

    wheels = 2

    def __init__(self, make, model):
        self.make = make
        self.model = model

        # Start in first gear.
        self.gear = 1;

    def change_gear(self, gear):
        self.gear = gear
```

We'll go through what's happening in each part of this code in more detail below, but briefly:

1. We define a new class called `Bicycle`.
2. A class attribute called `wheels` is defined and assigned a value.
3. The `__init__` method sets initial values for instance attributes called `make`, `model` and `gear`.
4. We define a method called `change_gear` which changes the value of the `gear` instance attribute.

To use this "blueprint", we call the class name in a similar way to if it were a function:

```
my_bike = Bicycle("Cannondale", "Optimo")
```

This line roughly translates to "use the `Bicycle` blueprint to create me a new object, which I'll refer to as `my_bike`". The `my_bike` *object*, also known as an *instance*, is the realised version of the `Bicycle` *class*. We can go ahead and create as many `Bicycle` objects as we like. There is still, however, only one `Bicycle` class, no matter how many instances of that class we create.

Attributes and methods

Classes define **attributes**, which capture the *properties* of that object. Objects can have attributes that are specific to that particular instance — for example, the currently selected gear — and attributes that are common to all instances of that class — e.g., the number of wheels for the Bicycle class. These are known as **instance attributes** and **class attributes**, respectively.

Taking the example code for the Bicycle class above, `wheels` is a class attribute, since it is

specified outside of a method,
whereas `make`, `model` and `gear` are instance attributes.

You use the `__init__` method to initialise (i.e. specify) an object's instance attributes by giving them their default value (or state). In the case of our `Bicycle` class, each bicycle object is initialised with a specific make and model.

To access the value of an attribute on a specific object, you use *attribute reference* syntax: `obj.name` — where `obj` is the name of the object (e.g. `my_bike`) and `name` is the name of the attribute (e.g. `model`). For example:

```
print(my_bike.make) # Cannondale
print(my_bike.gear) # 1
print(my_bike.wheels) # 2
```

Similarly, you can assign values to object attributes using the same syntax:

```
my_bike.model = 'Topstone'
my_bike.gear = 2
print(my_bike.model) # Topstone
print(my_bike.gear) # 2
```

Note that instance and class attributes are accessed in the same way; however, class attributes can also be accessed directly from the class, whereas instance attributes can't (because they are only defined for a given instance):

```
print(Bicycle.wheels) # 2
print(Bicycle.make) # AttributeError: type object
'Bicycle' has no attribute 'make'
```

Beware of mutating class attributes!

Many languages insist that the value of class variables can never change, but Python doesn't have this restriction. In Python you you are free to re-assign or mutate class variables as much as you like. Be careful though, because it can lead to some surprising behaviour and introduce some tricky bugs. Your code will normally be cleaner and easier to understand if you keep class variables constant.

Let's take a look at an example. In each case have a think about what you would expect to happen? Were you right?

```
class MyClass:
    mutableClassVariable = ['Hello']

obj1 = MyClass()
obj2 = MyClass()

# Re-assign the class variable on the class
MyClass.mutableClassVariable.append('World')
print(obj1.mutableClassVariable) # ['Hello', 'World']
print(obj2.mutableClassVariable) # ['Hello', 'World']
print(MyObject.mutableClassVariable) # ['Hello',
'World']

# OR, instead of the above, we re-assign the class
variable via the instance.
obj1.mutableClassVariable = ['Hello', 'World']
print(obj1.mutableClassVariable) # ['Hello', 'World']
print(obj2.mutableClassVariable) # ['Hello']
print(MyClass.mutableClassVariable) # ['Hello']

# OR, instead of either of the above, we mutate the
class variable (works on the class or the instance).
obj1.mutableClassVariable.append('World')
print(obj1.mutableClassVariable) # ['Hello', 'World']
print(obj2.mutableClassVariable) # ['Hello', 'World']
print(MyClass.mutableClassVariable) # ['Hello',
'World']
```

At first sight it looks very odd. Why does it make a difference if you mutate the value rather than re-assign it? Why is it different if you change the instance vs changing the class? How come changing one instance can also change the value for a different instance?

To figure all that out we'll need a deeper understanding of how variables and classes work in Python.

Firstly, remember that all variables in python are really just 'references' to a particular value. Multiple variables can reference the exact same value. This means that if you mutate a value, then all variables that reference that value will also change. By contrast, if we 're-assign' a variable that tells python to change our variable so that it references a new value. As a result, this doesn't impact any other variables.

Secondly, we should note that everything in python is really just an object. So our instances (`obj1` and `obj2`) are both objects of type `MyClass`, but the class `MyClass` is also an just an object. Every instance of a class contains a reference to this class object. You can see it just calling `obj1.__class__`.

Finally, what actually happens when you call `obj1.myVariable`? Python will looks for the value of `myVariable` in several different places in order - picking the first matching thing it finds. Those places are:

1. Some edge cases we are going to ignore for now
2. Looks in `obj1.__dict__` (i.e. looks through all the attributes for the instance)
3. Looks in `obj1.__class__.dict` (ie looks through all the attributes for the class)
4. If all else fails, throw an AttributeError.

OK - can we make sense of it now? See if you can piece together what happens.

1. When re-assigning the class variable on the class object, we update the value for everyone. This is because both instances reference the same class object, so if we change that, we affect everything.
2. When re-assigning the class variable via the instance, what actually happens is that python sets a new instance attribute with the same name. This new attribute hides the original one (we find a matching attribute at step 2, so never get to step 3)
3. When mutating the class variable, all the references are untouched - so when we change the value that everything refers to, we effectively update all the variables.

Hopefully this all makes a little more sense now and you know a little bit more about Python... but hopefully you also agree that this is a bit confusing. Good, clean code should be simple and easy to understand, so be wary when mutating or re-assigning class variables.

Classes also define **methods**, which describe the *behaviour* of that object. In the case of the Bicycle class above, it defines a `change_gear` method that allows you to change the currently selected gear on a bicycle object. Methods are declared similarly to functions, but take `self` as their first parameter. More specifically, these functions are known as *instance methods* because they are called on instances of the class, not the class itself. When calling an object's methods, you omit the `self` argument. For example:

```
my_bike.change_gear(3)
```

Notice how we only specified a value for `gear`. Python takes care of passing the value of `self` for you.

If we look back at the code for the `change_gear` method when we defined the `Bicycle` class, we can see that this method accepts the new gear value as an argument and sets the instance attribute `gear` to that new value. We can check to confirm that it has indeed been updated:

```
print(my_bike.gear)  # 3
```

Initialising objects with `__init__`

When we called `Bicycle("Cannondale", "Optimo")` before to create an instance, under the hood it is the class's `__init__` method that is being called. This method is responsible for setting up the initial attributes of the object. You can define a class without specifying an `__init__` method, in which case it will create an instance without any of its

Adding attributes and methods to objects

The most common way of adding attributes and methods to objects is to do the following:

```
class Tea(object):
    def __init__(self, type, ingredients):
        # constructor
        self.type = type
        self.ingredients = ingredients

    def prepare_tea(self):
        # use self.ingredients to make the tea
        (...)
```

However, we can also add attributes and methods in a dynamic way, by using setattr (https://python-reference.readthedocs.io/en/latest/docs/functions/setattr.html). It can be an existing one or a new one:

```
breakfast_tea = Tea("breakfast_tea",
["black_tea", ...])
setattr(breakfast_tea, "type", "blueberries
tea")
setattr(breakfast_tea, "preparation_time", 20)
```

Or even change existing methods...

```
def prepare_sweet_tea(self):
    # add a lot of sugar
    # use self.ingredients to make the tea
    (...)
setattr(breakfast_tea, "prepare_tea",
prepare_sweet_tea) # only for the given object
setattr(Tea, "prepare_tea", prepare_sweet_tea) #
for the class
```

However, "with great power comes great responsibility". So much freedom can end up with:

- security vulnerabilities
- bugs due to the change of class methods behavior or internal attributes in a way not expected by the class

attributes set. In the case of our `Bicycle` class, the `make` and `model` attributes are set when initialising the object, and are required parameters. You might be wondering what `self` means — well, wonder no more, because we're going to talk about it next...

The meaning of `self`

What's up with all the references to `self`? Notice how it appears as the first parameter to all of the `Bicycle` methods? What is it?

Why, it's the instance itself! Put another way, a method like `change_gear` defines the behaviour of changing gear for bicycles in general. Calling `my_bike.change_gear(3)` applies those instructions to the `my_bike` instance.

So when we write `def change_gear(self, gear)`, we're saying, "here's how to change gear on a Bicycle object (which we'll call self) to the specified gear (which we'll call `gear`). `self` is the instance of Bicycle that `change_gear` is being called on. That's not just an analogy, either: `my_bike.change_gear(3)` is just shorthand for `Bicycle.change_gear(my_bike, 3)`, which is perfectly valid (if rarely seen) code.

Inheritance

While object-oriented programming is useful as a modelling tool, it's true power starts to be realised through the concept of **inheritance**. Inheritance is the process by which a "child" class derives the properties and behaviour of a "parent" class. That might sound rather vague, so let's illustrate the concept with a specific example (albeit a rather tortured one!)...

Consider Vehicles'R'Us, a dealership that sells all types of vehicles, from motorbikes to trucks. Vehicles'R'Us pride themselves on their competitive pricing. Specifically, how they determine the price of a vehicle in their showroom: £5000

The “super” keyword

When inheriting from a base class, you'll often end up reusing the same method names as the base class, the most common example being the `__init__` function.
When this happens, how do you call the base class method from child class?
Python solves this problem by providing the `super()` method, which allows access to any of the base classes methods:

```
class Rectangle:
    def __init__(self, length,
width):
        self.length = length
        self.width = width

class Square(Rectangle):
    def __init__(self, length):
        super().__init__(length,
length)
```

You can also use `super()` to access base methods from further down the inheritance hierarchy by providing the target class as a parameter:

```
class ColouredSquare(Square):
    def __init__(self, length,
colour):
        self.colour = colour
        super(Rectangle).__init_
_(length, length) # Calls
__init__ from Rectangle and not
Square
```

multiplied by the number of wheels the vehicle has. They also have a very attractive buy-back scheme as well, offering a flat rate minus 10% of the miles driven on the vehicle. For trucks, that rate is £10,000; for cars, £8000; and for motorbikes, £4000.

If we wanted to create a sales system for this dealership using object-oriented techniques, how would we do so? What would the objects be? We would probably have a Sale class, a Customer class, an Inventory class, and so on, but we'd almost certainly have a Car, Truck, and Motorcycle class as well.

What would these classes look like? Here's a possible implementation of the Car class:

```
class Car:

    def __init__(self, wheels, make,
model, miles):
        self.wheels = wheels
        self.make = make
        self.model = model
        self.miles = miles

    def sale_price(self):
        return 5000.0 * self.wheels

    def purchase_price(self):
        return 8000.0 - (0.1 *
self.miles)
```

In reality, we'd likely have a number of other methods on the class, but let's focus on two that are of particular interest to us: `sale_price` and `purchase_price`. We'll see why these are important shortly.

Similarly, our Truck class might look something like this:

```
class Truck:

    def __init__(self, wheels, make,
model, miles):
        self.wheels = wheels
        self.make = make
        self.model = model
        self.miles = miles

    def sale_price(self):
        return 5000.0 * self.wheels

    def purchase_price(self):
        return 10000.0 - (0.1 *
self.miles)
```

But look — that's almost identical to the Car class! One of the most important rules of programming (in general, not just when dealing with objects) is "DRY" or "Don't Repeat Yourself". We've definitely repeated ourselves here. In fact, the Car and Truck classes differ by only a couple of characters (in the `purchase_price` method).

Our main problem is that we raced straight to the concrete: Cars and Trucks are real things, tangible objects that make intuitive sense as classes. However, they share so much data and functionality that it seems there must be an *abstraction* we can introduce here. Indeed there is: the notion of Vehicles.

A Vehicle is not a real-world object. Rather, it is a concept that some real-world objects (like cars, trucks and motorcycles) embody. We would like to use the fact that each of these objects can be considered a vehicle to remove repeated code. We can do that by creating a Vehicle class:

```
class Vehicle:

    base_sale_price = 0

    def __init__(self, wheels, make,
model, miles):
        self.wheels = wheels
        self.make = make
        self.model = model
        self.miles = miles

    def sale_price(self):
        return 5000.0 * self.wheels

    def purchase_price(self):
        return self.base_sale_price -
(0.1 * self.miles)
```

Now we can make the Car and Truck classes ***inherit*** from the Vehicle class, e.g.

```
class Car(Vehicle):
    ...
```

Here, the class in parentheses on the first line of a class definition specifies the class that it inherits from.

We can now define Car and Truck in a much more straightforward way:

```
class Car(Vehicle):

    def __init__(self, wheels, make,
model, miles):
        self.base_sale_price = 8000
        self.wheels = wheels
        self.make = make
        self.model = model
```

Mixin it up with multiple inheritance

It is possible for a Python class to inherit from multiple base classes:

```
class Engine:
    def __init__(self, engine_name):
        self.engine = engine_name

class Wheels:
    def __init__(self, number_of_wheels):
        self.number_of_wheels= number_of_wheels

class Car(Engine, Wheels):
    def __init__(self, engine_name,
number_of_wheels):
        super(Engine).__init__(engine_name)
        super(Wheels).__init__(number_of_wheels)
```

When inheriting from multiple base classes the base classes are often referred to as mixins instead (as they are mixed-into the class).

This can come in handy when you want to break down a class into multiple reusable components. For example I could now define a Bicycle class using just the Wheels mixin:

```
class Bicycle(Wheels):
    def __init__(self, number_of_wheels):
        super(Wheels).__init__(number_of_wheels)
```

However if we'd defined Car using a Vehicle base class like so:

```
class Vehicle:
    def __init__(self, engine_name,
number_of_wheels):
        self.engine = engine_name
        self.number_of_wheels= number_of_wheels

class Car(Vehicle):
    def __init__(self, engine_name,
number_of_wheels):
        super().__init__(engine_name,
number_of_wheels)
```

Then this would no longer be possible (as a bicycle does not have an engine).

```
        self.miles = miles


class Truck(Vehicle):

    def __init__(self, wheels, make, model, miles):
        self.base_sale_price = 10000
        self.wheels = wheels
        self.make = make
        self.model = model
        self.miles = miles
```

Abstract classes

The above code works, but still has a few problems. First, we're still repeating a lot of code. We'd ultimately like to get rid of all repetition. Second, and more problematically, we've introduced the Vehicle class — but should we really allow people to create Vehicle objects (as opposed to Cars or Trucks)? A Vehicle is just an abstract concept, not a real thing, so what does it mean to say the following:

```
vehicle = Vehicle(4, 'Honda',
'Civic', 0)
print vehicle.purchase_price()
```

A Vehicle doesn't have a meaningful `base_sale_price`, only its child classes Car and Truck do. The issue is that Vehicle should really be an **abstract base class** (ABC). These are classes that are only meant to be inherited from; you can't create an instance of an ABC. That means that, if Vehicle is an ABC, the following is not allowed:

```
vehicle = Vehicle(4, 'Honda',
'Civic', 0)
```

It makes sense to disallow this, as we never meant for Vehicles to be used directly. We just wanted to use it to abstract away some common properties and behaviour. So how do we make a class into an abstract base class? Simple! Python's `abc` module contains a class called ABC. Making a class inherit from ABC and making one of its methods virtual will make that class an ABC. (A **virtual method** is one that the ABC says must exist in child classes, but doesn't necessarily actually implement.) For example, the Vehicle class may be defined as follows:

```
from abc import ABC, abstractmethod

class Vehicle(ABC):

    base_sale_price = 0

    def sale_price(self):
        return 5000.0 * self.wheels

    def purchase_price(self):
```

Wrapping up functions: Decorators

Sometimes there's a need for performing a common operator after executing a function; for example, printing out to console that the function is complete:

```
import datetime

def say_hello():
    print("Hello")
    print(func.__name__ + " was called at " +
datetime.datetime.now().strftime("%m/%d/%Y,
%H:%M:%S"))
say_hello()
```

This can become tedious when you have to do this repeatedly for many functions.
Wouldn't it be nice if you could define the logging in one place and attach it to your functions as needed?
This is where decorators come in. By defining a decorator function:

```
import datetime

def log_function_call(func):
    def with_logging(*args, **kwargs):
        func(*args, **kwargs)
        print(func.__name__ + " was called at
" + datetime.datetime.now().strftime("%m/%d/
%Y, %H:%M:%S"))
    return with_logging
```

You can now add the decorator directly to your functions with the @ symbol:

```
@log_function_call
def say_hello():
    print("Hello")

say_hello()
```

Why does this work?
Well, the python interpreter actually interprets this as:

```
def say_hello():
    print("Hello")

say_hello = log_function_call(say_hello)

say_hello()
```

You can go as far as adding methods to classes with decorators:

```
class A(object):
    def __init__(self):
        self.x = 0

def add_method(cls):
    def decorator(func):
        setattr(cls, func.__name__, func)
        return func
    return decorator

@add_method(A)
def foo(self):
    print(self.x)

a = A()

# should print 0
a.foo()
```

Please see this link (https://book.pythontips.com/en/latest/decorators.html) for more on how to use decorators (such as adding decorator arguments, more usage examples and why @wraps should normally be used when writing a decorator).

```
        return self.base_sale_price - (0.1 * self.miles)

    @abstractmethod
    def vehicle_type(self):
        pass
```

Now, since `vehicle_type` has been marked as an abstract method, we can't directly create an instance of Vehicle:

```
vehicle = Vehicle() # Raises the following error
TypeError: Can't instantiate abstract class Vehicle with abstract methods
vehicle_type
```

As long as Car and Truck inherit from Vehicle and define the `vehicle_type` method, we can instantiate those classes just fine.

Returning to the repetition in our Car and Truck classes, let see if we can't remove that by moving up the common functionality to the base class, Vehicle:

```
from abc import ABC, abstractmethod

class Vehicle(ABC):

    base_sale_price = 0
    wheels = 0

    def __init__(self, make, model, miles):
        self.make = make
        self.model = model
        self.miles = miles

    def sale_price(self):
        return 5000.0 * self.wheels

    def purchase_price(self):
        return self.base_sale_price - (0.1 * self.miles)

    @abstractmethod
    def vehicle_type(self):
        pass
```

The '`@property`' decorator

A good practice in Object Oriented Programming is the use of getters and setters over the attributes. These allow us to have a little more control over what can be done with the attributes of a class - for example, in the code below we use a the `set_age` function to restrict the age attribute so that it can't be below 0.

```
class Student(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def get_age(self):
        return self.age

    def set_age(self, age):
        if age < 0:
            raise ValueError("Age below
0 is not possible")
        self.age = age
```

However, adding these methods to existing classes means changes all over the code. Here it comes into play @property decorator:

```
class Student(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @property
    def age(self):
        print("Get Value")
        return self._age

    @age.setter
    def age(self, value):
        print("Set Value")
        if value< 0:
            raise ValueError("Age below
0 is not possible")
        self._age = value

st = Student("Mark", 20)
# Set Value

print(st.age)
# Get Value
# 20

st.age = 25
# Set Value
```

In the above example, even though we do not change the way external code interacts with age attribute, under the hood python calls the two methods we just implemented to get and set the attribute.

Now the Car and Truck classes can be simplified to:

```
class Car(Vehicle):

    base_sale_price = 8000
    wheels = 4

    def vehicle_type(self):
        return 'car'


class Truck(Vehicle):

    base_sale_price = 10000
    wheels = 4

    def vehicle_type(self):
        return 'truck'
```

This fits perfectly with our intuition: as far as our system is concerned, the only difference between a car and truck is the base sale price. Defining a Motorcycle class then becomes similarly simple:

```
class Motorcycle(Vehicle):

    base_sale_price = 4000
    wheels = 2

    def vehicle_type(self):
        return 'motorcycle'
```

We've now managed to move all the common functionality for a vehicle to an abstract base class, leaving our concrete classes to focus on what makes them *different* from each other.

Unlimited arguments: Variadics

It is possible for functions to support an arbitrary list of arguments:

```
def test_var_args(farg, *args): # Note the
asterisk
    print("formal arg:", farg)
    for arg in args:
        print("another arg:", arg)

test_var_args(1, "two", 3)

# formal arg: 1
# another arg: two
# another arg: 3
```

You can even support unlimited keyword arguments:

```
def test_var_kwargs(farg, **kwargs): # Note the
double asterisks
    print("formal arg: ", farg)
    for key in kwargs:
        print("another keyword arg: {}:
{}".format(key, kwargs[key]))

test_var_kwargs(farg=1, myarg2="two", myarg3=3)

# formal arg: 1
# another keyword arg: myarg2: two
# another keyword arg: myarg3: 3
```

Why this is useful when you could just pass in a list or a dictionary manually instead?
Two reasons to consider are:

- It's often more expressive to use arguments directly than to create collections to wrap them and immediately unwrap them in the function
- This enables the construction of wrapper functions/ decorators that can handle all possible combinations of function arguments (as per the sidebar on decorators)

Chapter 4

Clean Code and Refactoring

Clean code

"Programs must be written for people to read, and only incidentally for machines to execute."

— Harold Abelson, *Structure and Interpretation of Computer Programs*

This frequently-quoted phrase underlies one of the core principles of *clean code*. But what is clean code, and what does that phrase really mean, anyway?

The idea behind it is that software programs are rarely, if ever, written once and never changed again. Code undergoes frequent changes and iterations, as bugs are fixed, features are added, and functionality is changed. The inevitable consequence of this is that code will have to be read — and more importantly, ***understood*** — by people in the future. Code that is readable and easy to understand makes their job much easier, and is therefore "better" than code that is undecipherable. Remember, too, that this "future person" might in fact be you!

While this quote slightly exaggerates the point (programs, by their nature, must also be executable by machines, after all!), it is a mantra that has helped to shape modern coding best practices.

What is *clean code*? And what makes code "clean"?

Readability, maintainability, testability, good design – they're all the same thing

The title says it all! Improving any of these elements will generally also improve the others. Good object-oriented design breaks a problem neatly down into classes, each of which is well encapsulated and tackles a single manageable concept. This in turn makes the code more readable and understandable, which makes it easier to maintain. When you have well encapsulated classes it is much easier to determine where changes need to be made and those changes should be required only in small number of well isolated places. This makes your changes smaller and safer, as it is much easier to understand any knock-on effects. We'll look more closely at testability in the next module, but small classes also tend to be easier to test, as its easier to work out how they should behave in any given situation. There are many synergies between these facets of coding; often you will find that attention to any one will yield improvements in the others. Keep them all in mind when you design, write and test your code!

Self-documenting, readable code

Messy code that was written months ago and is difficult to understand can cause a massive headache for developers, slowing down the team, increasing costs and making it harder to maintain quality at a high level. And it's just

no fun. Readability goes beyond just the minutiae of the line-by-line code, though. Try to keep the following higher level guidelines in mind when writing your code.

- Chose simple solutions over clever but harder to understand ones, even if they are a bit more verbose
- Use consistent and descriptive names for your classes, functions and variables. Try to ensure that these names match the terminology used in the business.
- Make your classes well encapsulated - with each class dealing with a single responsibility and exposing a straightforward API.
- Keep your code loosely coupled - so that each part/class can be thought about in isolation, without needing to understand the details of other parts/ classes (See the Law of Demeter)
- Follow existing conventions. If there's a common way of doing something in your project / team / industry then its usually sensible to follow it.

If you follow all these guidelines, you should find that your code needs very few comments in order to be clearly understood. If you find yourself wanting to add a comment, first ask yourself 'could I refactor this code so that it is clear enough to not need a comment?'. Often you will find that you can. That doesn't mean you shouldn't ever write comments though - they are still a very useful tool - but when it is immediately clear what the code does, we can use the comments to explain why we chose to do it that way. This is helpful in cases where we need to do something a little surprising for some reason - eg a bug in a library we are using forces us into a strange work-around.

One concept or level of abstraction per class or method (a.k.a. the Single Responsibility Principle)

Large classes and methods are difficult to deal with in many respects. You should be able to keep most methods to 10 lines or less by breaking them apart into more methods where necessary. By naming each method carefully the code will be self-documenting to a large extent. This contrasts with the popular tactic of having quite long methods (hundreds of lines long in the worst cases) with a comment every few lines explaining each chunk of code. Instead, each method's name can take the place of the comments and if there is more to say then that can be added in comments for each method.

Consider the following function, which might have several lines of code after each comment (omitted here for brevity) — but imagine each "..." is 2–10 lines of code:

```
def start_processing() {
    # Update the UI to show progress
bar.
    ...
```

```
    # Get the data to process.
    ...

    # Process the data.
    ...

    # Finished.
    ...
}
```

It would be better written as follows, using concisely-named small functions that are self-documenting and each individually very easy to read and understand:

```
def start_processing() {
    show_progress_bar()

process_data(get_data_to_process())
    hide_progress_bar()
}

...
```

Methods become especially difficult to follow if they contain a lot of nested logic — ifs, elses, loops, try-catch blocks, etc. Instead, try to keep only a very small amount of logic in each method and represent your overall logic with well-named methods calling each other. Each method should be quite easy to understand in isolation and ideally explains itself purely by virtue of its name and that of its arguments.

Splitting methods apart to cover complex logic introduces a potential problem though when there are a lot of variables in play. This was easy within a massive mega-method as all the variables were available to all the logic at once, but passing them as arguments through a series of small methods quickly leads to argument overload and the sort of confusing mess we were trying to avoid in the first place! What to do? One tactic is to introduce a 'context' class that simply packages all the data up so it can be passed around the methods without needing large numbers of arguments. Another is to break the logic itself out into a new class that maintains the data as member variables of the class rather than as local variables in methods. Use a new instance of this class, suitably populated with data each time you need to employ its logic. This leads to a greater number of smaller classes, each of which covers a smaller remit and is more easily understood, modified and unit tested.

Tell, don't ask

A common cause of code which fails the Law of Demeter test is taking a data-centric, rather than function-centric, view of your classes. You should aim to structure your code so that classes offer functionality, rather than exposing data and expecting callers to manipulate it.

For example:

`company.employees.add(new_employee)`

versus:

`company.add_employee(new_employee)`

- The latter makes it completely clear what functionality is supported by the Company class

    - this functionality can be easily and fully tested in isolation
    - it's easy to mock this functionality when testing other classes
- It completely hides the implementation detail of the company object
    - We can change how employees are stored within the company object without making any changes to calling code
    - We can adapt to changes in the new employee process by making localised changes to one method in the Company class
    - By not exposing `employees` directly, the scope for other classes re-using that property in unintended ways is much reduced. It's not possible to re-use the `add_employee()` method in unintended ways since it serves such a specific well-defined purpose, but exposing a list of employees is ripe for unexpected usages. It might seem like this is increasing flexibility, but it's probably just opening the door to much tighter coupling between classes. It's much better for your class to describe its specific capabilities via public methods than to expose its implementation details by giving broad access to its underlying data.

Refactoring

Refactoring has been described by Martin Fowler as "the art of safely improving the design of existing code". Refactoring is therefore a process of source code transformation — and of continuous improvement. The main focus of a refactoring should be restructuring the code so that its meaning is obvious to a new reader. Refactoring should take place in several cases:

- Before writing new, more complex features
- When copying over/reusing code
- When you find that fixing one thing breaks multiple other components
- Whenever you judge that the code has become complex and hard to read (for a new reader or even for the writer themselves)

Benefits

- readability for other developers (especially new ones) or even the writer themselves after some time passed
- maintainability — makes it easier to maintain and upgrade the functionality of the code, resulting in fewer bugs
- less development time in the future — a badly written code generates overheads over time, that will impact the number of hours a task will take

Techniques and tips

- Look for multiple lines doing the same thing — try to convert to a loop, function, or parent class.
- Make small, incremental changes, rather than one big change — it's easier to backtrack if you get stuck, and more likely to highlight better ways to re-write the code as you go along.
- Break up complex conditional statements — move the conditional expression to a new function and give it a meaningful name.

```
# initial code
if event.day == "tuesday" and (event.month == "June"
or event.month == "July" or event.month == "August")
and 8 <= event.hour <= 12:
     # serve special summer breakfast
elif event.month == "December":
     # closed, Marry Christmas!
else:
     # serve normal menu
```

```
# refactored code 1
# break the if conditions into functions
def is_special_breakfast_day(event):
     special_breakfast_days = ["tuesday"]
     return event.day in special_breakfast_days

def is_summer(event):
     summer_months = ["June", "July", "August"]
     return event.month in summer_months

def is_special_breakfast_interval(event):
     interval_start = 8
     interval_end = 12
     return interval_start < event.hour <
interval_end

def is_holiday(event):
     holiday_months = ["December"]
     return event.month in holiday_months
```

Lambdas

Lambdas provide a way of declaring simple functions inline without having to write a full function definition:

```
a_variable = lambda a, b: a + b
print(a_variable(2,3)) # 5
```

A lambda declaration contains (in order):

1. The keyword lambda
2. A list of function arguments (keyword arguments are allowed)
3. An expression

The expression becomes the return value of the function. Note that you cannot:

- Use multiple expressions within a lambda
- Use return or flow control (if, while)
- Assign the lambda a function name
- Use type hints for the arguments or the return type

Apart from these restrictions lambdas act like any other function object. You can even immediately invoke them:

```
print((lambda a, b: a + b)(2,3)) #
5
```

Because of their limitations Lambdas are best used when:

- They are relatively straightforward (and their usage is easy to understand)
- They are not going to be heavily re-used

If not, they should be promoted to full functions.

```
(...)
# make small changes and make sure the code still works
if is_special_breakfast_day(event) and is_summer(event) and
is_special_breakfast_interval(event):
     # serve special summer breakfast
elif is_holiday(event):
     # closed, Marry Christmas!
else:
     # serve normal menu
```

```
# refactored code 2
# eliminate event parameter by using a class
class EventWrapper:
     # these are all static variables now
     special_breakfast_days = ["tuesday"]

     summer_months = ["June", "July", "August"]
     holiday_months = ["December"]

     interval_start = 8
     interval_end = 12

     def __init__(self, event):
          self.event = event

     def is_special_breakfast_day(self):
          return self.event.day in self.special_breakfast_days

     def is_summer(self):
          return self.event.month in self.summer_months

     def is_special_breakfast_interval(self):
          return self.interval_start < self.event.hour < self.interval_end

     def is_holiday(self):
          return self.event.month in self.holiday_months

     def should_serve_special_breakfast(self):
          return self.is_special_breakfast_day() and self.is_summer() and
self.is_special_breakfast_interval()

(...)
wrappedEvent = EventWrapper(event)
if wrappedEvent.should_serve_special_breakfast():
     # serve special summer breakfast
elif wrappedEvent.is_holiday():
     # closed, Merry Christmas!
```

```
else:
     # serve normal menu
```

Tech Debt

Tech debt is a concept that describes the result of actions that prioritize faster delivery over code quality. Just like any financial debt, the technical one has an "interest" that accumulates in time (in general, the code is harder to extend) and the code needs to be refactored at some point. The main reason why tech debt appears is meeting a deadline (choosing a limited solution over a more apropiate, time consuming solution).

Tech debt management

- track the tech debt — create cards / keep a record of the tech debt
- Boy scout rule — leave the code better than it was or fixing some issues as part of other tasks, if possible
- negotiate a tech debt budget with the client to be used for fixing tech debts
- create metrics to track the priority and urgency of the tech debts — cost of the fix versus impact of the issue (client losing money, sensitive data, etc) and probability of the issue to happen
- review the tech debt backlog from time to time
- convince the client the importance of fixing tech debt — might be the hardest one, as one needs to understand the client's priorities and make their case based on the added value / risk mitigation (which in general is far from obvious when it comes to tech debt: "The app is working, why should I invest money in something with no visible effect?")

Chapter 5

Trunk-based development

Merge Conflicts

One of the trickiest parts of collaborative source control is the handling of merge conflicts. This can happen when developers make changes to the same files on different branches and then want to merge the two branches together. At this point someone needs to make an executive decision on how to handle the conflict.

Often these conflicts will be quite easy to resolve, but that won't always be the case. Major conflicts can cause major headaches for teams as:

- They can be very time consuming to fix
- They increase the risk of introducing bugs while you fix them
- Its easy to accidentally delete newly added features while fixing conflicts

Luckily, there are lots of strategies for keeping conflicts small and manageable.

Keep Your Changes Small

Targeted changes that are small in scope generally have far fewer lines of code. As a result, you are less likely to have any conflicts. Even if you do have conflicts, it is much more likely that they will be small and easy to fix.

Merge frequently

The longer you are working on a branch, the more likely it is that whatever on the main branch will have changed, causing conflicts that you will need to sort out. The more often you can get your work back on the main branch, the easier it will be for you and your team.

Encapsulate your Code

As well as making the code easier to understand and maintain, isolated, well encapsulated classes make it easier to make your changes in a small number of places. This will make merging much easier, as it makes it more likely that your work will be isolated from the work your teammates, and therefore free of conflicts.

Communicate with your team

If you have a good idea what your teammates are working on, its is often possible to spot issues before they happen. If you are about to start some work that is likely to clash with work somebody else is doing, then go and talk to them! Perhaps its better to wait for them to merge first? If not, it should at least be helpful to understand the changes they are making.

Pair

It is well known (though often forgotten) that doubling the number of developers on a team rarely halves the amount of time it takes to finish a project. One reason for this is that the more changes that are happening at once, the harder it is to avoid conflicts. One way to reduce this issue is to pair. This should reduce the number of separate 'streams' of work and reduce the amount of time it takes

to complete each piece of work, allowing you to merge more frequently.

Git Workflows

We have now got a better idea about the types of changes we'd like to make, but how does this work in practice? What branches should we create? When do I merge them? What gets released and how does that work?

The short answer is that it should depend on your team. There are a lot of different possible workflows all with their own pros and cons. What's important is that you find one that works well for your team, and then continue to refine and improve it as your team develops. Here are a few popular options that may make good starting points.

Trunk-Based Development (aka Mainline)

Trunk-Based Development is often held up as the gold standard that everyone should be aiming for. It relies on several things all coming together, but can provide great results. The idea is that you have a single branch (often called `master` or `trunk`) and all changes are applied directly to this branch. The workflow is:

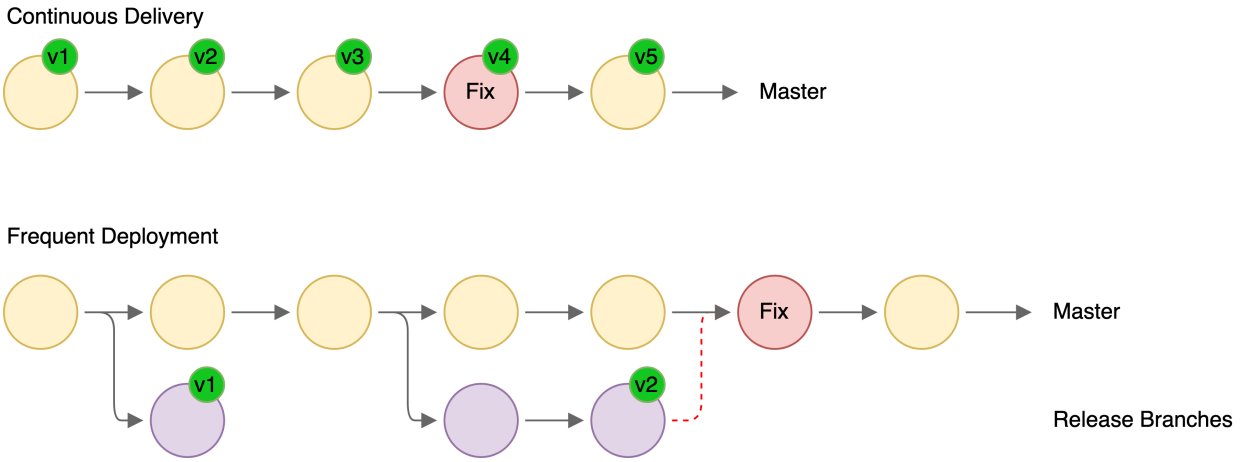
1. Pick up a new task
2. Check out the most recent version of the 'master' branch
3. Make the smallest change that helps move the task forward (without breaking things)
4. (Often) Code goes through some kind of review process
5. Code is merged into the master branch
6. (Recommended) Some automated tests run to make sure everything still works
7. Repeat from 2 until the task is complete.

The key points are that there is only ever a single long-lived branch (master/trunk) and all the developers are committing to it as often as possible (ideally many times each day). This means that all the changes are small, our master branch is always up to date with recent code from all the developers, and we never have to do a big merge that might cause nasty conflicts.

Releases

Often, teams that practice Trunk-Based Development will also practice Continuous Delivery (releasing their code to users after every change to the codebase). If you can do this, then congratulations - many teams consider this to be the ideal workflow.

Teams that don't practice Continuous Delivery will often release by creating a 'release' branch from master. Any sign off or manual checks can happen against this branch, and if any bugs are found, the fixes are applied to the 'master' branch and then cherry-picked onto the 'release' branch. Once ready, the code can be deployed from the release branch.



When it works well

Trunk-Based Development works great with experienced developers working in a small team that aims to release very frequently. If you have an environment like this then Trunk Based Development allows you to almost entirely avoid the issue of merges, requires very little overhead to manage and allows you to develop features rapidly using a series of small, safe changes which are quickly delivered to your users.

Potential Drawbacks

There are of course some drawbacks. Committing directly to master makes it easy to accidentally break the master build. When this happens, development is delayed as it is difficult for new changes to be added until this is fixed. Trunk-Based Development therefore requires your developers to be disciplined in what commits they push, and quick to solve any issues that crop up.

Trunk-Based Development also becomes a lot more difficult when the release process is slow. Both Continuous Delivery and 'cherry-picking' fixes back into the release branch rely on a quick release cycle. Without it, Continuous Delivery becomes painfully slow, and cherry-picking becomes dangerous as the release branch gets more and more out of date.

Perhaps the biggest downside though is that it is sometimes difficult to break tasks down into small commits that can each be safely merged and released without breaking existing code at any point. One strategy to help with this is Feature Toggles which we'll talk about in more detail a little later. These allow us to put code in place which doesn't get run in our production environment. This is exactly what we need to continue with Trunk-Based Development, but does cause added complexity in the code, which would not otherwise be required.

GitHub Flow

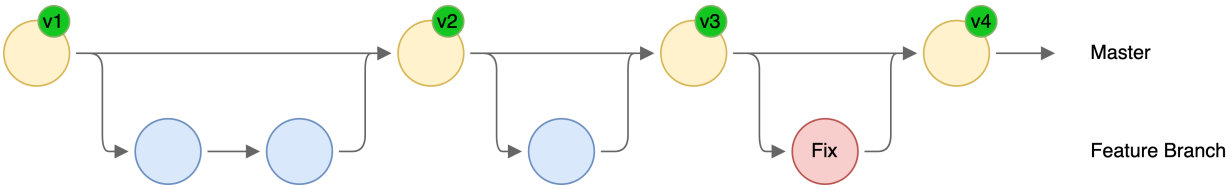
GitHub Flow was (unsurprisingly) made popular by GitHub. You can use similar methods though with whatever tools your team uses. It is similar to Trunk-Based Development, but allows short-lived branches alongside our long-lived master branch. Allowing these reduces some of the downsides from the Trunk-Based approach we talked about before at the expense of requiring us to deal with a few more merge conflicts. The workflow is:

1. Pick up a new task
2. Check out the most recent version of the master branch
3. Create a new 'feature' branch for the task you are working on
4. Commit changes to this branch
5. Automated tests run on this branch
6. A Pull Request (PR) is raised, and a team member reviews the code
7. Any markups etc are done on this branch
8. Code is merged into the master branch
9. Automated tests run again to check everything still works
10. Repeat!

Again the trick is to get merging as quickly as possible. In practice, this means making sure that each task/feature is small or splitting up a larger task into multiple PRs.

Releases

Exactly the same as with Trunk-Based Development, teams can either practice Continuous Development or split off short lived release branches.



When it works well

GitHub Flow makes it really easy to setup (and enforce) that automated checks run and code reviews happen **before** any code gets merged into the master branch. This extra step helps to ensure that whatever is on master is in a good state and is rarely broken. As a result of this extra structure, many teams feel that this workflow is a little more forgiving than Trunk-Based Development and a little easier especially for junior developers or larger teams.

As it allows for slightly larger pieces of work, it also reduces the need for Feature Toggles or code that doesn't yet get run. You are likely to still need them - but a little less often than with Trunk-Based Development.

Potential Drawbacks

The drawback compared with Trunk-Based Development is that the time between commits to master is typically longer. Even though the branches are in theory short lived and therefore quite easy to handle, most teams struggle to avoid a few larger and longer-lived pull requests creeping into their workflow. This is difficult to avoid entirely, and as soon as it happens merge conflicts inevitably follow!

Git Flow

Git Flow is another popular approach but it is more complex that the other options listed here. We would only recommend it if you are sure that something simpler can't work for your team.

Instead of having just a single master branch, Git Flow sets up multiple long-lived branches and then gives you strategies for managing these and merging code between them. Typically, you will have a 'master' branch that keeps track of code that has been deployed to production, and a 'develop' branch which represents the most up to date code that has not yet been released. The workflow then works identically to the GitHub Flow case, except that we are committing to the 'develop' branch not the 'master' one.

Releases

Git Flow expects you to not have very frequent releases. If you want Continuous Delivery, this isn't going to be the right workflow to pick.

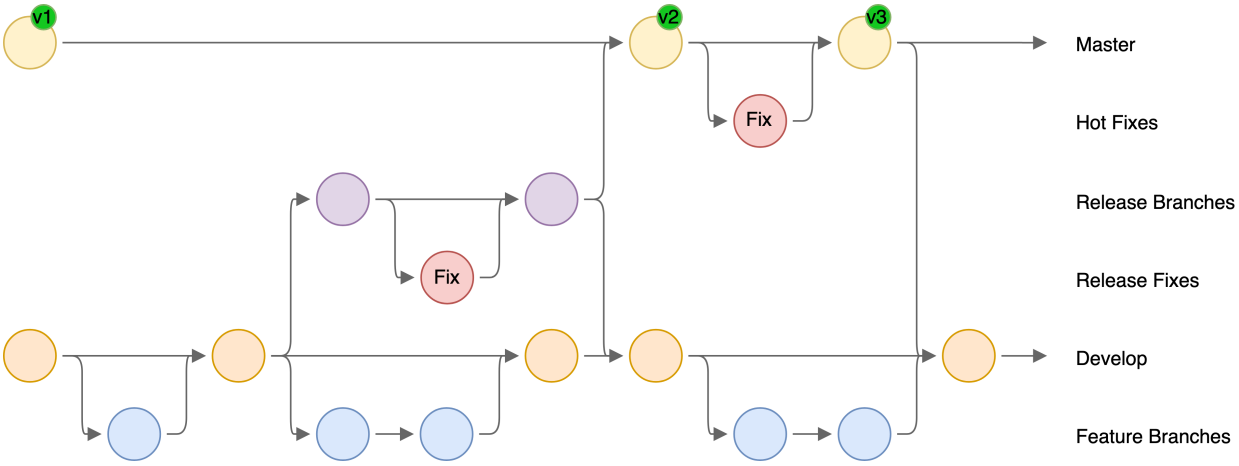
Once the developers are happy with the code for the release, we follow the following release workflow:

1. Create a new 'release' branch.
2. Carry out any sign off / testing that is required. This time though, once the release branch is created, we never cherry-pick / merge further code from develop.
3. If we need any fixes:
    1. Create a new branch off the release branch
    2. Add commits to fix the bug
    3. Raise a PR to merge the bug fix branch into release
4. Once the release is ready, raise a PR and merge the release branch into the master branch
5. Deploy the code to production
6. Merge the master branch back into develop (so that develop now has those bug fixes)

Hot Fixes

Git Flow also defines a special procedure for fixing bugs in production. If we spot a bug that we'd like to fix, we don't want to go through the normal release process again,

because that would involve releasing more new code that hasn't been signed off yet. So instead we perform a hot fix release. This works similar to how we fix the release branch. We first create a new branch off master, then fix the bug on that branch, before raising a PR to merge back to master. We can then deploy the fix, and finally merge master back into develop again.



When it works well

If your release process is very slow, you release rarely and you can't fix those things then you will find yourself needing longer lived branches to handle releases. In situations such as this, you may find that a process such as Git Flow helps to unify the team around a documented 'way to do things' and makes it easier to manage the merges and pain points involved in maintaining multiple long-lived branches.

Potential Drawbacks

Releasing code and handling bug fixes is much more complex with Git Flow than the other processes. You will find that changes require more thought, more Pull Requests and more time than with other approaches. It is also much trickier to explain to new members of the team.

Having multiple long lived branches also greatly increases your changes of nasty merge conflicts. For example, lets say you add a hot fix to master. You then need to merge master back into develop - but develop may have moved on by now, leading to some surprisingly annoying merge conflicts (that can often only be sorted out by the repository Admin).

Feature Flags/Toggles

One concern with committing changes frequently to the master branch is the impact this has on the release process:

- Just because the CI/CD checks pass, this doesn't mean the end product provides a reasonable or consistent UX
- There might be incomplete features with "Not Implemented Yet" blocks of code that shouldn't be hit/shown in production
- The security implications of new changes may not have been reviewed or pen tested

Normally addressing these issues would be very time consuming for each release, making the release process slow and risky

Feature flags (or toggles) are a way of locking incomplete or new functionality away as needed so that people managing the release can feel more confident that regressions will not occur.

Typically this is implemented by gatekeeping *in progress* or *new* features behind flow control statements (if/switch etc.) that change flow base on flags (i.e. config) that can be managed before/after release

- This requires greater discipline by developers as they need to ensure features are fully ring-fenced
- In particular, integration tests need to work for all reasonable permutations of possible configuration flags
- Flags should also be cleaned/removed once their corresponding features have been released and fully adopted

The Trunk Based Development website has its own section covering feature flags.

Branch By Abstraction

Feature flags help keep the release process running smoothly when merging *in progress* or *new* features into the master branch. However, it can be difficult to apply this approach directly when making large changes to existing functionality. You cannot simply "turn off" functionality while you work on changing it, and the old functionality may be tightly coupled to the rest of your code, leaving you with no obvious place to put a feature toggle in your application code.

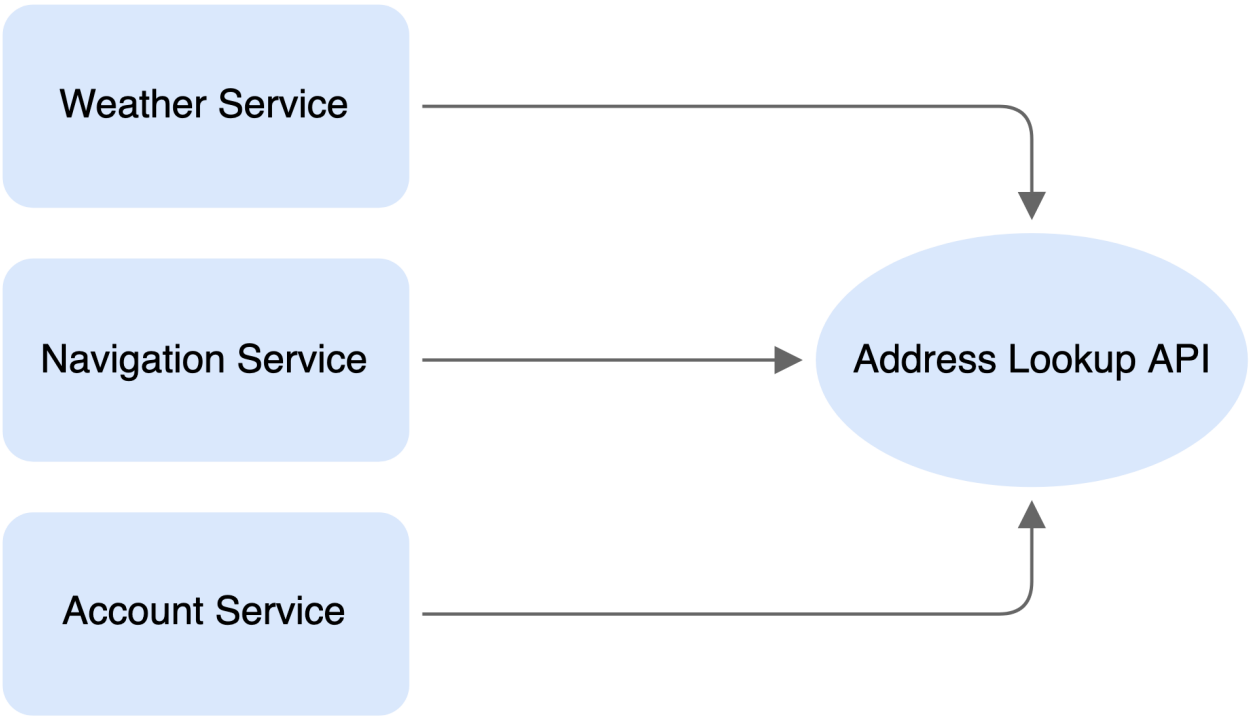
Branching by abstraction enables developers to rework existing functionality while reducing the risk of broken deployments and without blocking other team members. It involves separating the to-be-replaced code from the rest of your application before making any changes to it.

The Trunk Based Development website has its own section covering branch by abstraction.

Step 1: Original Code

Before making any changes your code might look something like the example below. Here we have a fictional web service, where different code components (the Weather Service, Travel Service and Account Service) all make calls to a common dependency. In the example it's called the Address Lookup API but the detail is unimportant - it could be anything.

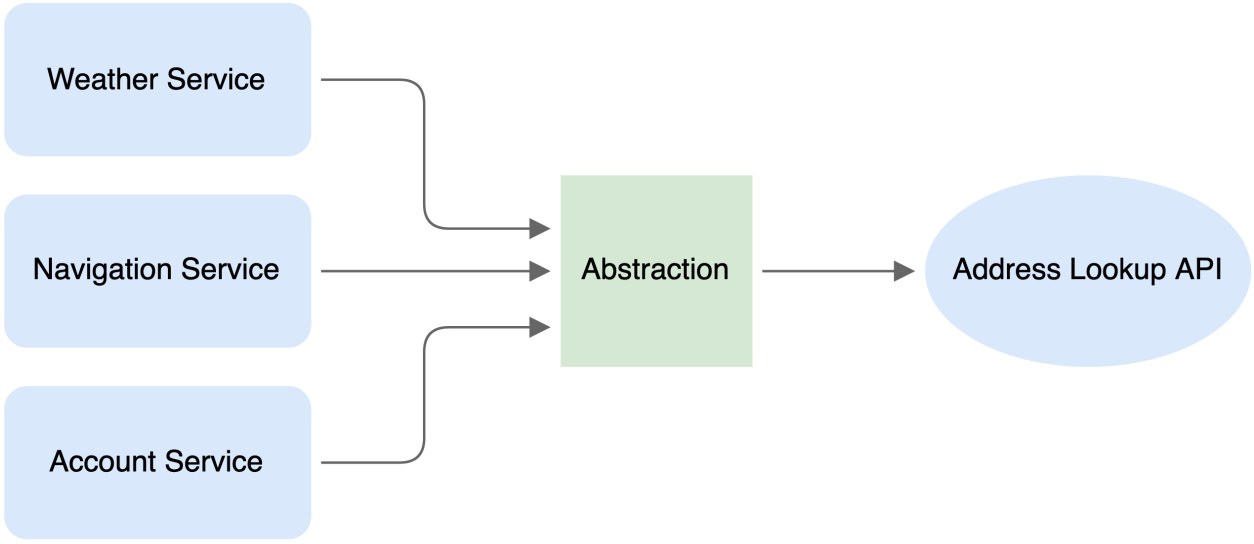
This example walks through how to replace "Address Lookup API" with a different component. Perhaps the old API is deprecated, or no longer viable for commercial reasons. Whatever the case, we want to remove it with the minimum disruption to the deployment process and the development process of our colleagues.



Step 2: Add Abstraction

To begin, add an abstraction layer between the old code ("Address Lookup API") and the client code that uses it. Depending on the nature of your work, abstractions could be internal to the application code - for example a new Python class. Alternatively, they could relate to infrastructure - for example adding a web proxy between two services. The branching by abstraction methodology helps in both cases.

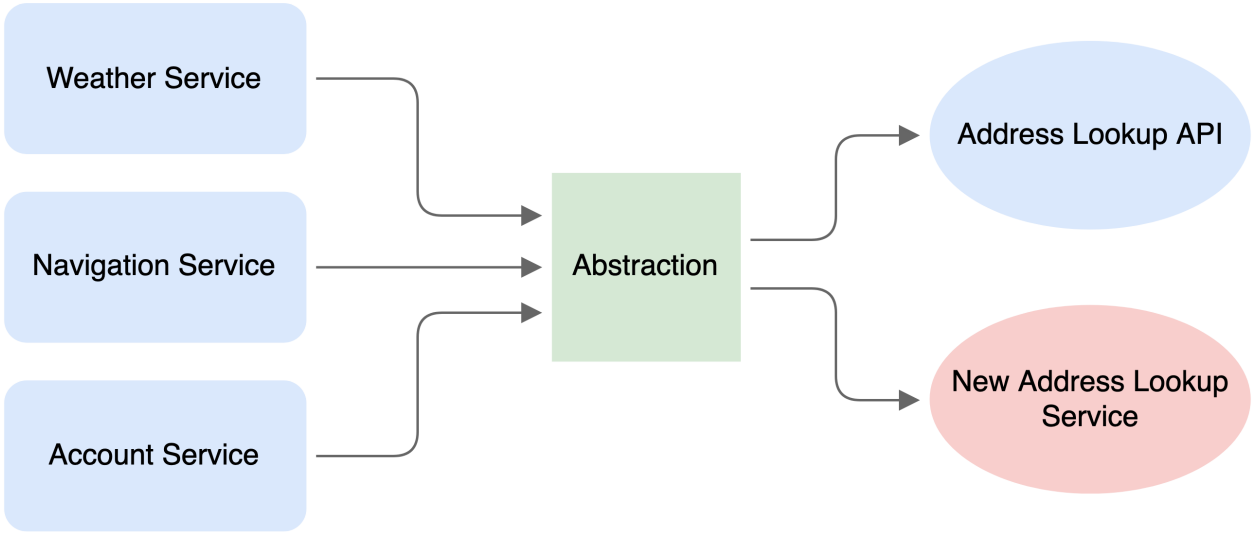
Merge the abstraction into the master branch without changing the existing Address Lookup API. This is a low-risk operation as we haven't changed any business logic.



Step 3: Add New Code

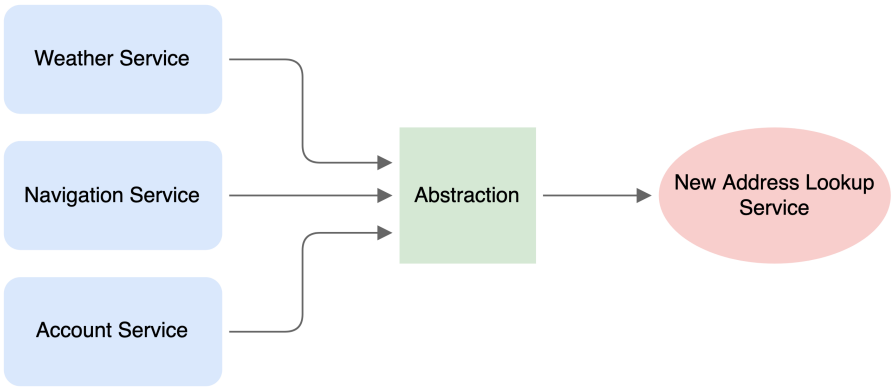
Add your new replacement code (the "New Address Lookup Service") and connect it to the abstraction alongside the old code. The abstraction should, by default, send all calls to the old code, but clients should have an option to use the replacement version.

Once deployed to the master branch, you can gradually switch features to use the new version (via the abstraction). This incremental approach mitigates the risk of serious service outages while you make a major code change by splitting it into multiple steps. Eventually, no code will use the old version.



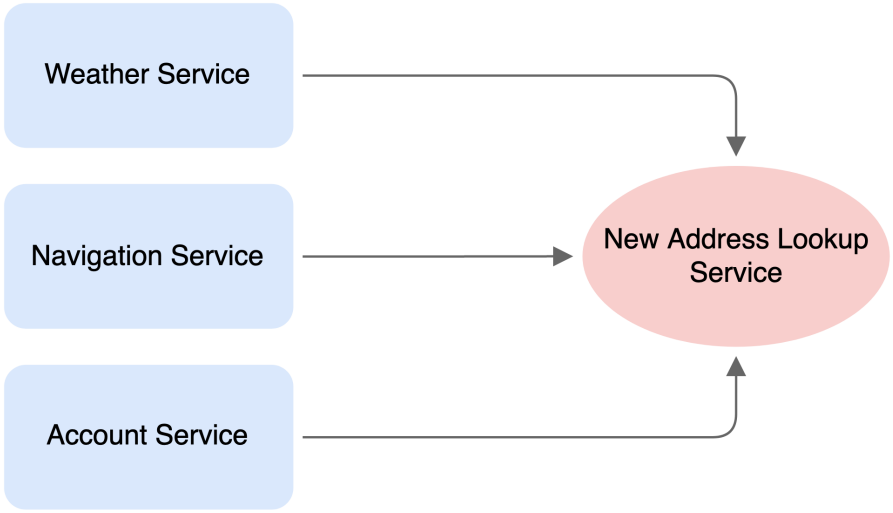
Step 4: Remove Old Code

Once all the client code is using the new dependency (New Address Lookup Service), you can safely remove the old version (Address Lookup API). This operation is safe because all client code is already using the new version.



(Optional) Step 5: Remove Abstraction

By this point all your code is running using the new dependency and the old one is gone. Congratulations, your feature switch is complete! You can now remove the abstraction and call the dependency directly from your client code. However, think carefully before doing this. Is the abstraction causing any harm? In many cases it is helpful to keep the abstraction. Abstractions act as interfaces between different components of your application. They encourage you to keep your code modular and follow the Single Responsibility Principle.



Always delete unused code

It may be tempting to leave the old code in place, just in case you need it in future. Don't do this! Keeping old code makes it harder to maintain a codebase, as developers can't be sure at a glance what code is in active use. It's much better to remove unused code as soon as possible to keep the repository in good shape. The old code is not lost - it can always be retrieved from the version control system (e.g. git) if needed in future.

Chapter 6

Additional Reading: APIs

Exercise: Marvel Comics API

Write a Python command line program that asks the user for the name of a Marvel character and then returns a short bio/description of that character.

Your program will need to:

- Ask for input from the user.
- Make HTTP requests to the Marvel Comics API.
- Process the JSON response from the API and print out relevant details.
- Handle the error case where no Marvel character is found that matches the user input.

The following resources should help get you started:

- Marvel's interactive API documentation
- Python `requests` library for making HTTP requests (**hint:** you'll need to install the package using pip before you can use it in your code)

Stretch goals

Possible extensions you could consider:

- Allow "starts with" or "fuzzy" searches.
- Provide summaries of all comics that the character appears in.

Authentication via OAuth 2.0 and OpenID Connect (OIDC)

We briefly mentioned both OAuth and OIDC in the section about authentication, but lets take a closer look at these two ideas and learn how to implement them in an application.

OAuth

It is frustrating having to create a new account for every application that you use online. Many people end up either using the same password for everything (which reduces security), or regularly forgetting passwords (which is frustrating). Wouldn't it be good if you could re-use a single account across multiple sites? OAuth's goal is to allow exactly that, in a controlled and secure manner.

In short, by using OAuth, a site can authenticate a user by just asking a 'provider' to do the authentication for it. This is great for the site, as it doesn't have to worry about doing authentication itself, and great for the user as they only need to remember their account details for a single 'provider' - not all of the individual sites. As a result use of OAuth has become quite common - there are now several providers (often large organisations like Google and Facebook) and many sites are happy to provide this sign in option (if you've ever seen a 'Sign in with Google' button this is how it works).

As you can see, when building an application, OAuth promises some large potential benefits:

- Popular with users (as they don't have to create or remember a new account just for your site)
- Authentication and account management are handled by the provider, so you don't have to! This means that you don't have to worry about a huge range of issues such as:
    - storing passwords
    - multi-factor authentication
    - password resets

However as usual, it isn't without its drawbacks!

- Not all users will have an account with all providers. To counter this, you can:
    - accept some users might not be able to use your site (not usually a great option)
    - add multiple providers
        - but this increases complexity and still doesn't guarantee that everybody will have one that works for them
    - also have another way to log in - eg password and username
        - but now you lose the benefit of not having to worry about doing your own authentication etc,

- and now you have 2 separate login systems which makes your application more complex

How it works

Note: There are 2 major versions of it, OAuth 1.0 and OAuth 2.0. We will be working with version 2 as its a little simpler to implement and more widely used. Version 1 is similar though if you ever need to use it.

Step 1 - Register your App

The details vary between providers, but they will all require you to register your app with them. Once registered, your app will be issued with a Client ID and a Client Secret. The ID can be public, but as the name suggest the secret should be kept secure. The combination of ID and Secret will be used later to authenticate the application (similar to a username and password).

Step 2 - Redirect users to your provider

Once you've registered with a provider, you can add that provider as a sign in option on your site. When a user selects this sign in option, your app should add some information about what your app is and what permissions it would like to have, and then redirects the user to the provider's site. The permissions your app asks for is called the 'scope'.

On the provider's site, the user will sign in, and then grant consent for your app to have that scope.

Step 3 - Parse the Provider's response

Once the user has signed in and granted consent, the provider will send a message to your application containing a 'code'. This works by your app providing a 'callback' endpoint which is called by the provider.

Step 4 - Get an access token

Your app can then make a new request to the provider to exchange this code for an 'access token'.

Code vs Access Token

What's the difference between that 'code' and the 'access token'? After all, they are just strings, and the code is only used to collect the access token... why couldn't the provider just return the access token straight away?

The reason is security. In order to be sure that it safe to issue an access token the provider needs 3 things:

- Confirmation that the user is who they claim to be
- Confirmation that the user is happy to provide access to the scope
- Confirmation that the application is what it claims to be

When the provider issues the 'code', it only has the first 2 confirmations. It is confident about who the user is and what the user wants, but it isn't yet sure that the application is what it claims to be. So instead of issuing an access key, it instead issues a code. The application can then make a separate request, including the 'code', the 'Client ID' and the 'Client Secret'. The code provides confirmation for the first 2 points, and the ID + Secret authenticate the application. The provider can now be sure that it is safe to issue the access key.

Step 5 - Use the API

Now we've got the access key, we can use it to make requests to the API.

Note that the access key will only provide access to information / actions that are defined in the scope that we requested back in step 2.

OpenID Connect (OIDC)

OpenID Connect is a built directly on top of OAuth. It's goal is to add a few extra concepts and standards to help make things easier. Most notably, it includes

- a 'user-info' endpoint which returns information about the authenticated user in a standard format.
- a 'discovery/configuration' endpoint - this allows OIDC clients to call just this single endpoint, and then automatically find out what other endpoints exist and how to use them.

Exercise

Lets put all of this into practice by adding OAuth with GitHub to an application. To get you started, we've already made a sample application for you. Start by forking this repo on GitHub and then following the instructions in the README to get everything set up.

Before we get going, let's have a quick look around the application. Here are some of the things you should know:

- Its a flask app, like you have seen before
- It uses flask-login - as the name suggests, this library helps with login! We are going to use it to set session cookies so that a user remains logged in across multiple requests.
- it uses oauthlib - this library is there to make it easier for us to make the OAuth and OIDC calls by doing a little of the heavy lifting for us.
- it uses SQLite at a database (don't worry too much about this - we'll talk about databases later on in the course)
- GitHub has an OAuth provider with good documentation. It also partially implements the OIDC specification - it has the `user-info` endpoint, but doesn't have a discovery endpoint, so we'll have to manually make the calls we'll need.

Now onto the task! Right now, the `login` endpoint and the `callback` endpoint are both empty. We are going to need to fill those in. The steps to get this working follow the same pattern as the steps above.

- Start by registering your app with a provider - we recommend GitHub, but you can pick something else if you like. You can find documentation here!
  - Add your new CLIENT_ID and CLIENT_SECRET to the `.env` file.

- Next take a look at the `login` endpoint. Its empty right now. Change it so that it instead redirects the user to GitHub's sign in endpoint, including the app's id and the scope we'll need. (docs here)
    - hint: take a look at the prepare_request_url method.
- We can now switch to looking at the callback method. We've got 4 things to do here
    1. Parse the code that's given to us by the request GitHub sends to us
    2. Use that code to obtain an Access Key (hint - take a look at the prepare_token_request and parse_request_body_response methods)
    3. Use the Access Key to call the user-info endpoint to find details of our user (hint - take a look at the add_token method)
    4. Construct a new user object, store it in your database and create a new session for them (most of this has been done for you)

Stretch goals

Possible extensions you could consider:

- Convert your program to a Flask web app that displays the character's details and thumbnail image (this is a bigger stretch goal!)

Chapter 7

Additional Reading: Type Hints

What are Type Hints?

Type Hints (also known as Type Annotations) allow you to specify the types for the variables, parameters and return values that you define.
For example, in this method, we have used the `: str` syntax to indicate that the 'name' parameter should be a string, and the `→ str` syntax to indicate that the return value should also be a string.

```
def greeting(name: str) -> str:
    return f"Hello {name}"
```

Adding these type hints doesn't affect how the code runs (python will ignore them), but they can be used by other tools to make it easier to write code.
For example, in many IDEs, if anyone tries to call this function with anything other than a string, the IDE will highlight this as a mistake.

```
def greeting(name: str) -> str:
    return f"Hello {name}"


greeting(5)
```

Expected type 'str', got 'int' instead

If you prefer command line tools, then projects such as mypi (mypy-lang.org) can be used to check for any type errors in your code base.

Why are they helpful?

Without type hints, it is up to the programmer to keep track of what type every variable is or could be. For small scripts and programs this is often not very difficult, but as the project gets larger it can get harder and harder to be sure that you have everything correct.
For example, without type hints:

- If you come across a method written by someone else (or by you a long time ago) can you be sure what types of parameters it accepts without opening it up and looking through the code?
- If you are editing a method, can you be sure what types the inputs will be? Is the method given a string every time, or is it sometimes something different? Can you be sure without checking through the whole codebase?

In older versions of Python, your only defence against these issues were to add comments indicating how a method should be used or to write automated tests which check that everything works the way it should.
Type Hints add another mechanism for solving these issues, and they do so in a concise manner that can be easily understood by developers and enforced by our tools. Good use of Type Hints can eliminate a whole class of possible errors and also increase the speed and ease of development.

Chapter 8

Additional Reading: Clean Code

SOLID principles

SOLID is a set of principles that underlie a lot of clean code and good object oriented design.

SOLID stands for:

- **S**ingle Responsibility Principle
- **O**pen-Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Each of these principles is worthy of further discussion.

Single Responsibility Principle

The Single Responsibility Principle states that each code module (for example, a class) should only be responsible for one aspect of the system's functionality / behaviour.

What does it mean to "be responsible for"? A helpful approach is to equate responsibility with "reason to change". Consider a class which produces VAT reports. There are (at least) two reasons this class might need to change (i.e. be edited or rewritten):

- The VAT rules change
- The reporting format changes

Two reasons for change means two responsibilities. Ideally the code would be restructured to separate these responsibilities into two classes, so that any one requirements change will only lead to changes in one class.

Why is this a useful guideline to follow? There are two key benefits:

- Code becomes shorter and more self-contained. This makes it easier to understand, and less likely to contain bugs.
- It's easier to find the code associated with some piece of functionality, e.g. if there is a bug or changes are required.

Open-Closed Principle

The Open-Closed Principle states:

software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification

In other words, it should be possible to add to the behaviour of a class without modifying the class itself.

By way of example, consider a `switch` statement. This consists of a number of cases, with different behaviour in each case. However, if you want to add a new case you need to make a change to the `switch` statement; this violates Open-Closed. A better model might be to implement a separate class for each case, inheriting from a common base class or interface; thus when you need a new case, you can just add a new class.

This doesn't mean we should plan on never changing any code we write - that would be excessive, and lead to messy codebases and significant extra development cost. However we should *predict what changes are likely to be needed*, and write our code accordingly. For example:

- A graphics package may support drawing many different shapes (squares, circles, etc.). It is highly likely that the range of shapes will be extended in future, so a good design will support adding new shapes without modifying the existing code. This might mean implementing shape-drawing via subclasses of a common base class; new subclasses can be added without needing to change the base class. This minimises the cost of adding new functionality and reduces the chances of bugs being introduced.
- An airline's booking website calculates airfares. It is very unlikely that this airline will diversify their business significantly into selling different types of service, and hence it is probably *not* worth building the calculation engine to allow calculating prices for different things (hotels, car hire, toasters...) without modification to the original code. Doing so would add a lot of extra cost and complexity without any likely future pay-off.

Further reading

The Open-Closed Principle is often a source of confusion, mostly because its name doesn't encapsulate the fact that you don't *always* want to "close your code to modification". There is a detailed analysis of this naming confusion here: https://codeblog.jonskeet.uk/2013/03/15/the-open-closed-principle-in-review/. A slightly more accessible, but less comprehensive, explanation is here: https://8thlight.com/blog/uncle-bob/2013/03/08/AnOpenAndClosedCase.html.

Liskov Substitution Principle

The Liskov Substitution Principle says that an instance of some class can be replaced with an instance of one of its subclasses, without breaking anything.

Object Oriented Programming is built on this principle - if you have a class `Apple` that derives from a parent class `Fruit`, then variables of type `Fruit` can take an `Apple` without any compilation errors.

However, merely satisfying the compiler is not sufficient to comply with the Liskov Substitution Principle. The *behaviour* of your code needs to meet the requirement too. For example, suppose your `Fruit` class has a method `Peel`, which peels the fruit. `Apple` supports peeling. But a second subclass `Grape` does not - grapes are not commonly peeled, so this subclass throws an exception when peeled. This is a violation of Liskov's principle - passing a grape into some code that relies on the peeling

behaviour of fruit will cause the program to fail.

In practical terms, the Liskov Substitution Principle means that when you create subclasses, you should make sure that they only build upon the behaviour of the parent class, and not change it in a way that could conceivably break code that relies on that parent's behaviour.

Interface Segregation Principle

The Interface Segregation Principle states that clients of a class should not be exposed to methods that they don't need.

Suppose you are building a class that handles reading and writing data to/from a CSV file. Consider some client code that needs to read the CSV file, but has no interest in writing. Should we expose the full set of read / write operations to this client? The Interface Segregation Principle says no - instead we should implement separate interfaces for reading and writing, and pass the reading interface to this particular client.

Why is the Interface Segregation Principle beneficial?

- The *intent* of the code author is clear. The client code receives an instance of the reading interface, and this clarifies that this code is only interested in reading data from it.
- The code is easier to change in future. You could pass in a class that reads from some other read-only data source, and the absence of methods to write data will not cause any problems.
- Coupling is reduced. Coupling means linking parts of your code together; in this case, the client code and the CSV writing logic. The more coupling you have the more complex your code is to analyse and understand (tending to make it more error prone), and more pragmatically the longer it will take to compile your code (e.g. any change to the CSV writer will cause recompilation of the client code mentioned above, even though it doesn't actually use that code). Using small interfaces to encapsulate the functionality we actually need minimises this coupling.

Dependency Inversion Principle

The Dependency Inversion Principle also aims to reduce coupling in your code. It has two parts:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Instead, details should depend on abstractions.

Consider the earlier example of a CSV reader / writer class, and some client code that uses it - say, to import financial records from a spreadsheet. The easiest implementation is for the "high-level module",

in this case the financial record importer, to create an instance of the "low-level module", the CSV reader. Hence the importer has a dependency on the reader.

The first step of implementing the Dependency Inversion Principle would be to give the CSV reader an interface (let's call it "data reader"), and have the financial importer depend on this interface instead. The classic approach is to use *Dependency Injection* - we pass an instance of the data reader interface into the importer (we "inject" the dependency), and hence the importer no longer needs to worry about the lower level module - we could swap out the CSV reader for some other data reader and the importer would continue to work fine.

The Dependency Inversion Principle goes one step further than this, however. When we created the data reader interface, we based it on the CSV reader class. The abstraction (the interface) depends on the details (the low-level module, the CSV reader). We probably have several other users of this data reader interface, and at some point we may realise that we need to provide a slightly different set of data reading operations - we will change the interface, and that means we'll also need to change our importer. A change to the low-level details has caused a change to the high-level modules - that's the wrong way round and violates the principle.

To address this, we must make the data reader interface a property of the *high-level* module, the importer. Let's create exactly the same interface, but call it "financial record reader". The CSV reader can still implement this interface with no changes. But because the interface has a more specific stated purpose, tied to the importer, we won't change it for unrelated reasons; if we do start overhauling how our CSV reader works, that's no reason to change the interface.

So, to fully follow the Dependency Inversion Principle we must inject dependencies into our classes, using interfaces that are "owned" by the class receiving the injection. The benefit of this is that changing one piece of code should not need to have a knock-on impact on code that depends upon it - a change only impacts code that the changed code depends upon.

Further reading

If you are familiar with the technique of Dependency Injection, there is a good explanation of why this doesn't go far enough to count as Dependency Inversion here: https://lostechies.com/derickbailey/2011/09/22/dependency-injection-is-not-the-same-as-the-dependency-inversion-principle/.