

Fix the code

```
n = 3
i = 0
while i < n:
    print("£")
    i -= 1
```

Change, add or remove **one** character
so that it outputs 3 “£” symbols

How many more ways can you do it? There are at least 4 ways!



**Corndel
Digital.**

in association with

Softwire

The Corndel DevOps Engineering Programme

in association with Softwire

Module 1 – General Purpose Programming Workshop

Agenda

1000 Welcome and introductions

1030 Part 1: Python Fundamentals

- Recap + New Stuff (30 min)
- Exercise (1 hour 30 min)

1230 Lunch Break (1 hour)

1330 Part 2: Version control with Git

- Recap + New Stuff (20 min)
- Exercise (50 min)

1440 Part 3: HTTP and web apps

- Recap + New Stuff (20 min)
- Exercise (1 hour)

How the Workshops Will Run

- Will typically involve a recap of core reading material
- Some new concepts required for the exercises will be introduced
- Exercises will typically be carried out in pairs/groups
- Workshop format may change slightly over time and depending on the module

Objectives of this Workshop

- Reinforce understanding of coding fundamentals in Python
- Build familiarity with common git commands, branching & raising Pull Requests
- Have an initial look at building web apps using Flask

Python Fundamentals

Part 1: Understanding Basic Python

Recap

You should already know this, but as a reminder:

How you assign variables

```
>>> my_name = 'John Smith'  
>>> my_age = 42
```

How you perform operations (such as arithmetic)

```
# Addition  
>>> six = 2 + 4  
  
# Subtraction  
>>> four = 6 - 2
```

- Basic Data Types, with operations allowed on each:

```
>>> first_string = 'Hello'
>>> second_string = 'World'
>>> first_string + second_string # 'HelloWorld'

>>> na = 'Na'
>>> batman = 'Batman'
>>> 8 * na # 'NaNaNaNaNaNaNaN'
>>> 8 * na + batman # 'NaNaNaNaNaNaNaNBatman'
```

```
>>> a_string = 'Example'

>>> a_string[0] # E
>>> a_string[1] # x
>>> a_string[6] # e

>>> a_string[-1] # e
>>> a_string[-3] # p
```

- Different types of flow in Python:

```
if username == 'user':
    print('Hello, User')
    if password == 'squirmbag':
        print('Access granted')
    else:
        print('Access denied')
```

```
times_run = 0
while times_run < 10:
    print('Hello!')
    times_run = times_run + 1
```


- Functions Definitions & Usages:

```
def print_item(name, price_in_pennies):  
    formatted_price = '£{:.2f}'.format(price_in_pennies / 100.0)  
    print('Item: ' + name)  
    print('Price: ' + formatted_price)  
  
print_item('Milk', 85)  
print_item('Coffee', 249)  
print_item('Orange Juice', 110)
```

- Complex Data Types (Such as Lists & Dictionaries):

```
>>> list_of_numbers = [3, 1, 4, 5, 7, 2, 6]  
>>> list_of_numbers.remove(4)  
>>> list_of_numbers # [3, 1, 5, 7, 2, 6]  
  
>>> list_with_repeats = [1, 2, 1, 3, 2]  
>>> list_with_repeats.remove(2)  
>>> list_with_repeats # [1, 1, 3, 2]  
>>> list_with_repeats.remove(1)  
>>> list_with_repeats # [1, 3, 2]
```

```
>>> favourite_colours = {}  
>>> favourite_colours['Alice'] = 'Purple'  
>>> favourite_colours['Bob'] = 'Green'  
>>> favourite_colours # {'Alice': 'Purple', 'Bob': 'Green'}
```

Part 1: New Concepts

Basic File Operations

Python provides various built-in functions for handling reading/writing of files:

```
f = open("test.txt", mode='w')  
f.write("my first file\n")  
f.write("This file\n")  
f.write("contains three lines")  
f.close()
```

```
with open("test.txt", 'r') as f:  
    text_string = f.read()  
    # Run f.close()
```

```
print(text_string)  
# my first file  
# This file  
# contains three lines
```

The 'mode' keyword argument let's us specify whether we're expecting to write ('w'), read ('r') or otherwise. [See the docs for more options](#)

It's good practice to explicitly close file handle after you're done with it – but it's easy to forget!

To simplify that, the 'with' syntax here will handle that for us; at the end of the block it will automatically close the file.

Tuples

Creation:

```
my_tuple = 1, 2, 3
my_tuple = (1, "Hello", 3.4)
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(my_tuple[0])
# "mouse"
```

Can be used to return multiple values from functions:

```
def divide_with_remainder(x, y):
    return x // y, x % y

output = divide_with_remainder(13, 5)
print(output)
# (2, 3)
value, remainder = divide_with_remainder(18, 7)
print(f"{value}, {remainder}")
# 2, 4
```

List Comprehensions In Python

Format:

```
[expression for item in list]
```

Simple Example:

```
h_letters = [ letter for letter in 'human' ]  
print(h_letters)  
# ['h', 'u', 'm', 'a', 'n']
```

More Complex Examples:

```
squared_odd_numbers = [ x*x for x in range(20) if x % 2 != 0 ]  
print(squared_odd_numbers )  
# [1, 9, 25, 49, 81, 121, 169, 225, 289, 361]
```

```
pythagorean_triples = [  
    (x, y) for x in range(1, 10)  
    for y in range(1, 10)  
    if math.sqrt(x*x + y*y).is_integer()  
]  
print(pythagorean_triples)  
# [(3, 4), (4, 3), (6, 8), (8, 6)]
```

Part 1: Exercise

The Ultimate Goal

The steps on the next slides will lead to a Python script that can process a file with a list of instructions:

```
goto 4
```

```
replace 1 2
```

```
remove 3
```

```
goto 2
```

```
goto calc x 3 5
```

```
replace 6 10
```

Step 1:

Write a basic Python "calculator".

It should accept 3 pieces of input from the user: a string that's one of "x", "+", "-", or "/" (an operation), an integer (parameter A), and another integer (parameter B).

It should then emit the result of performing the operation on A and B.

For example, if your application asks the user for an operation and 2 numbers, and the user enters "+", "1", "2", then the application should output "3".

If the user supplied "/", "5", "2", the application should output "2.5".

If the user supplied "x", "5", "0", the application should output 0.

Step 2:

Next process the following file: [Link](#)

Each line contains a calculation statements prefixed by "calc":

```
calc x 2 5  
calc / 10 5
```

Create a new Python script, and within it:

- Read in the new file (see the earlier slide!)
- Compute the value of each line using the approach from step 1
- Add up the results from all the lines and send the results to the trainer

Hint 1: For reading the lines from the file you may want to use `file.read().splitlines()` to build a list of lines.

Hint 2: you may want to use `string.split()` to break up the parts of each calc line.

Step 3 (Page 1 of 2):

Next navigate the following file: [Link](#)

This has goto statements like the following

```
goto 27
```

This means go to line 27 in the file and read the statement there. Please note that calc and goto statements can be combined like so:

```
goto calc / 27 9
```

This is equivalent to goto 3.

For simplicity assume that we cannot nest calc statements, decimals are rounded down and out of bounds gotos (i.e. invalid line numbers) do not occur.

Step 3 (Page 2 of 2):

Starting from line 1, use the rules above to navigate the document, stopping when you've **hit a statement you've seen once before (they are allowed to be from different lines!)**.

When finished please send the statement and line number the code has stopped on to a tutor – but feel free to carry on and start the next stage while you wait for confirmation.

Step 4 (Page 1 of 2):

Finally navigate the following file: [Link](#)

This has some additional statements.

The goal is to process the file, starting from line 1, stopping when you've **hit a statement you've seen before or manage to jump outside the file by a goto.**

When finished, please send the line number & statement to the trainer to confirm.

Step 4 (Page 2 of 2):

Additional Statements:

```
remove {line_number}
```

- Remove line {line_number} from the file (if the line number does not exist do nothing) and then
- Read the next instruction after this remove statement

```
replace {line_number_1} {line_number_2}
```

- Replace line {line_number_1} with line {line_number_2} (if either line number does not exist do nothing) and then
- Read the next instruction after this statement

Lunch

Back at 13:30

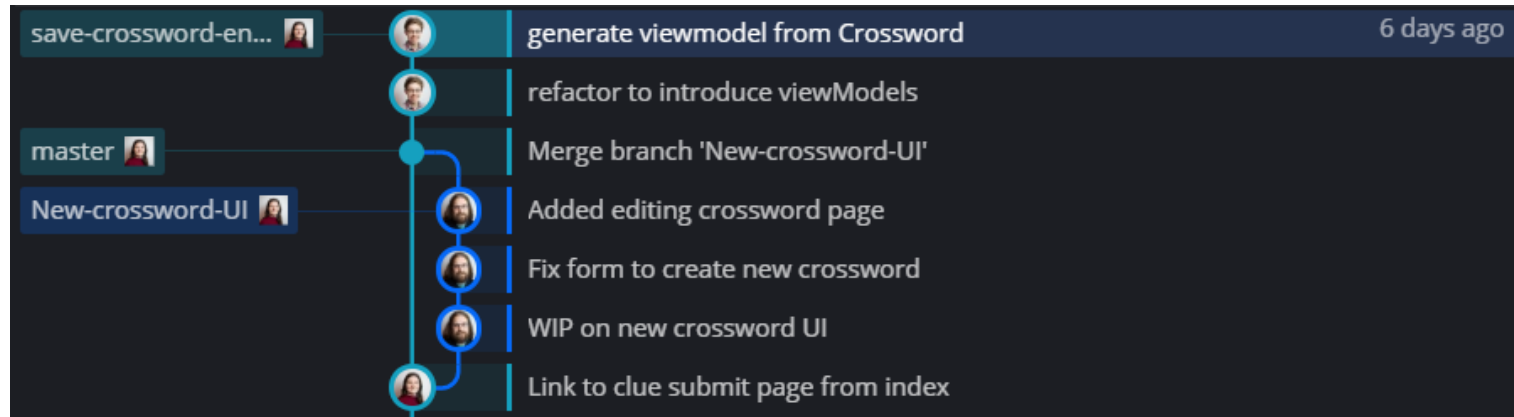


Part 2: Git & Source Control

Part 2: Version Control In Git

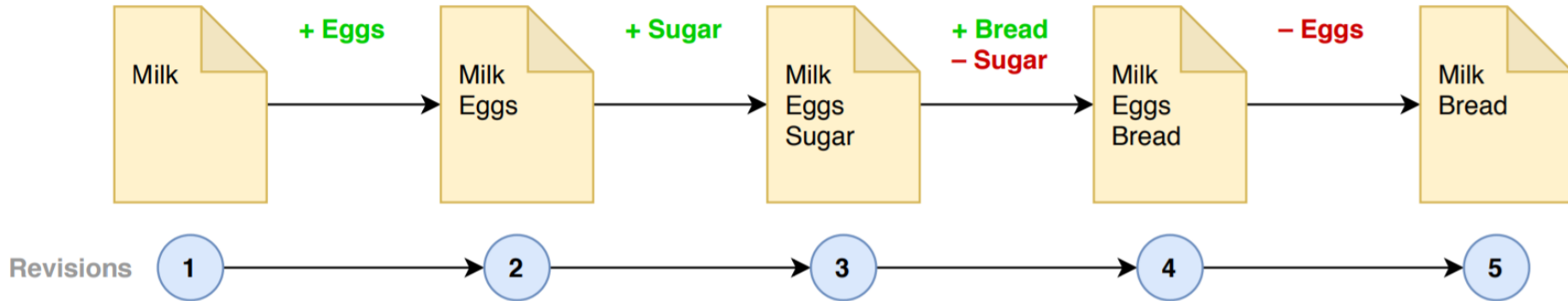
Purpose of Git

Git is a version control system that allows you to track changes to your codebase over time, examine the differences between the current state of your codebase and the last known working version, as well as check out your codebase as it existed at various points in time:



Commits

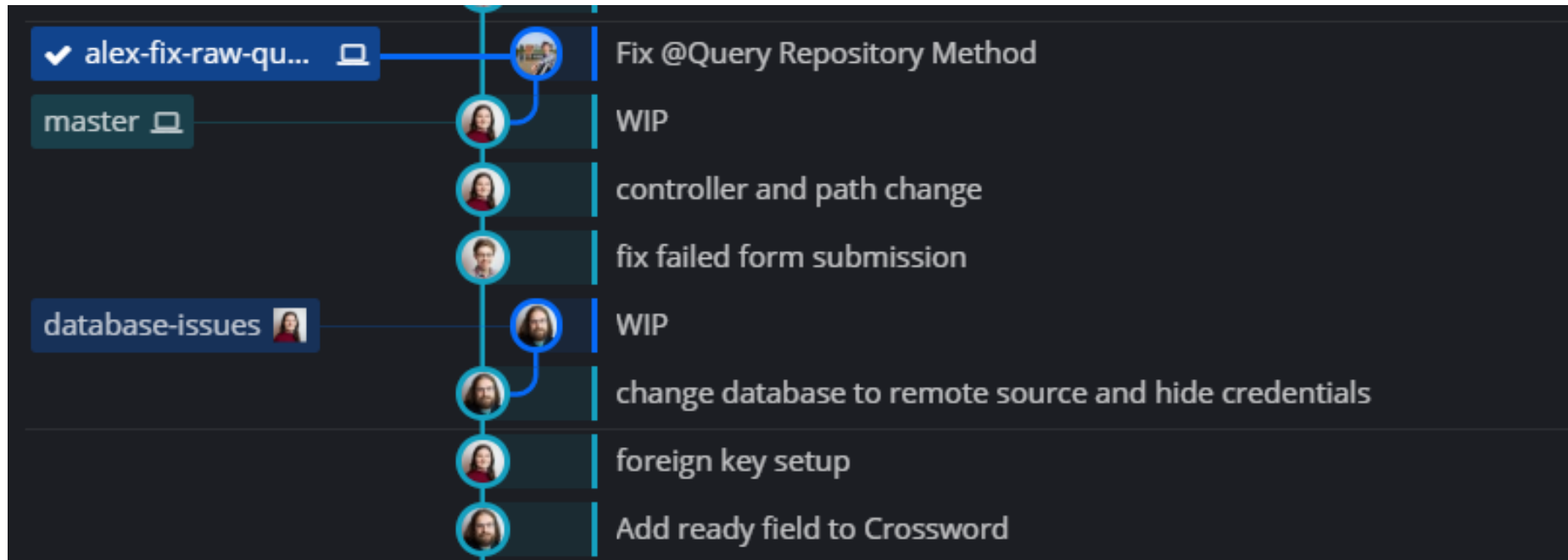
"Commits" are snapshots of your code that you've saved (like working on an important document and having copies called "My doc V1.docx", "My doc V2.docx", "My doc V3.docx", etc).



Every time you make a commit, you build on the previous one, making a chain of commits in a sort of timeline, retaining the ability to "time travel" to any point in time.

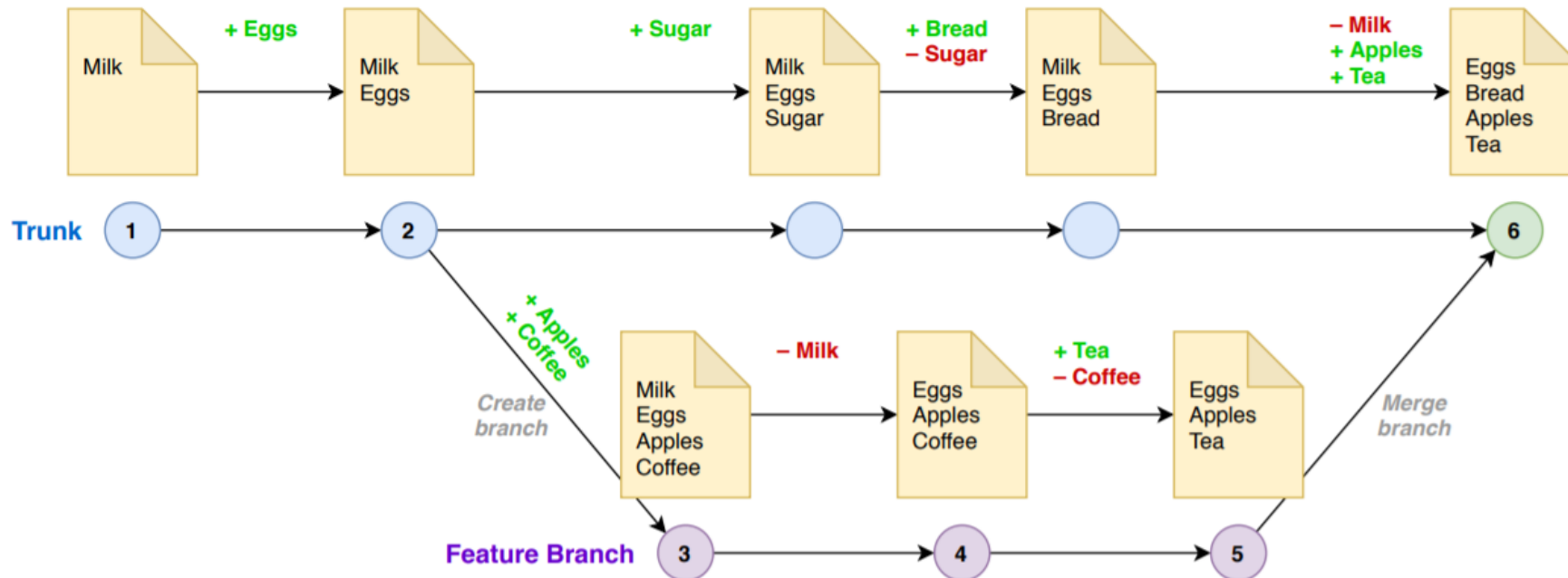
Branches

You can also make "Branches" - parallel universes in which your codebase looks slightly different. You can travel to different branches too.



Merging Of Branches

Branches can be merged into each other (typically creating a “merge commit”)



Basic Git Commands

Here is a summary of all the git commands you need to know about right now:

```
$ git status
$ git init
$ git add <path to file>
$ git commit -m "<commit message>"
$ git branch <new branch name>
$ git checkout <target branch>
$ git remote add <remote name> <remote url>
$ git fetch
$ git pull
$ git push
$ git merge <branch to be merged into current branch>
```

Part 2: Exercise

The Goal

Setting up a GitHub repository with your code from part 1.

You will:

- Create a local repository, with some initial commits and branches
- Create an online public repository and push up code from your local repository
- (Extension) Try resetting & reverting
- (Extension) Create some merge requests for your repository
- (Extension) Create (!) and resolve a merge conflict

Step 1

From the directory of your code from part 1, initialise a git repository and commit everything you have as an initial commit.

- If you don't have git on your machine at this point please download Git from <https://git-scm.com/downloads>
 - a) Add a readme file (called README.md) that describes how to use your code (in simple plaintext) and make a new commit with a message "Added README"
 - b) Add a text file with the name NewFile.txt to the directory of your local repository and create an appropriate commit.

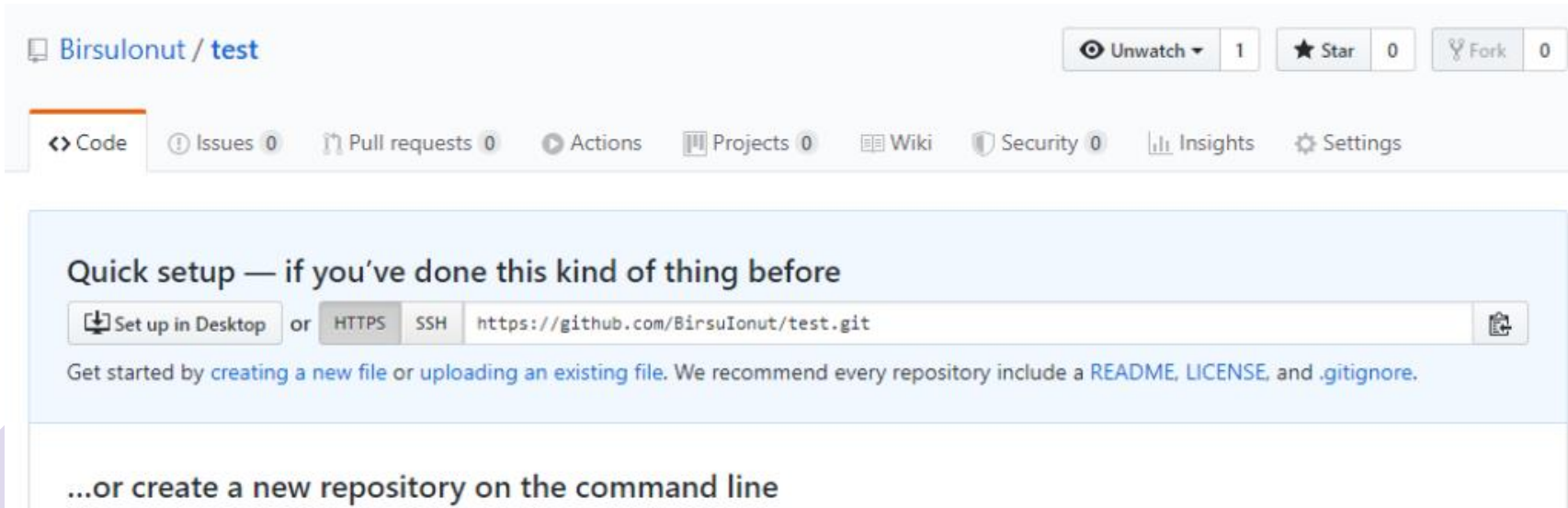
You can ask a trainer to review the state of your git repository if you'd like to confirm before proceeding to step 2.

Tip: You may want to run a git visualiser tool like gitk or GitKraken to track your commits/branches

Step 2 (Page 1 of 2)

Next, let's push up the code from parts 1 & 2 to a remote repository:

- If you don't have an account already, create one on <https://github.com>
 - You will want to create a personal one if you only have a work account
- [Create a new repository](#), giving it a name, making the repo public and leaving the README and .gitignore options unselected



The screenshot shows the GitHub interface for a repository named 'test' by user 'Birsulonut'. At the top, there are buttons for 'Unwatch' (1), 'Star' (0), and 'Fork' (0). Below this is a navigation bar with links to 'Code', 'Issues' (0), 'Pull requests' (0), 'Actions', 'Projects' (0), 'Wiki', 'Security' (0), 'Insights', and 'Settings'. The main content area has a light blue header with the text 'Quick setup — if you've done this kind of thing before'. Below this header, there are three buttons: 'Set up in Desktop', 'HTTPS', and 'SSH'. The 'SSH' button is selected, and the text 'https://github.com/BirsuIonut/test.git' is displayed in a text box. To the right of the text box is a copy icon. Below the text box, there is a paragraph of text: 'Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).' At the bottom of the screenshot, there is a section titled '...or create a new repository on the command line'.

Step 2 (Page 2 of 2)

- d) Follow the instructions under “... **or push an existing repository from the command line**”
- e) You will be prompted to provide your GitHub username and password in the terminal
 - If you prefer not to enter your credentials over a terminal you can create a SSH key to use with your GitHub account. This will involve running:

```
$ ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
```

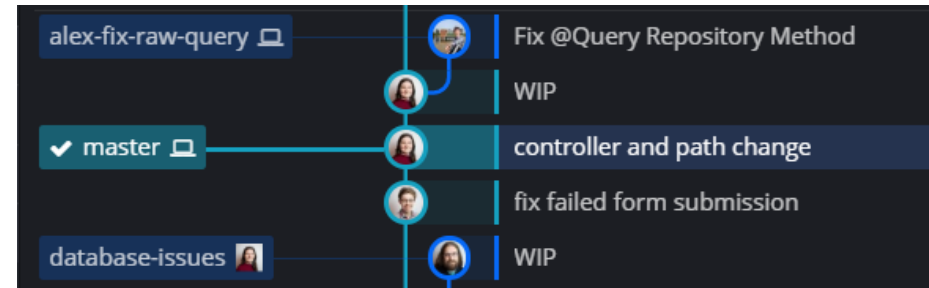
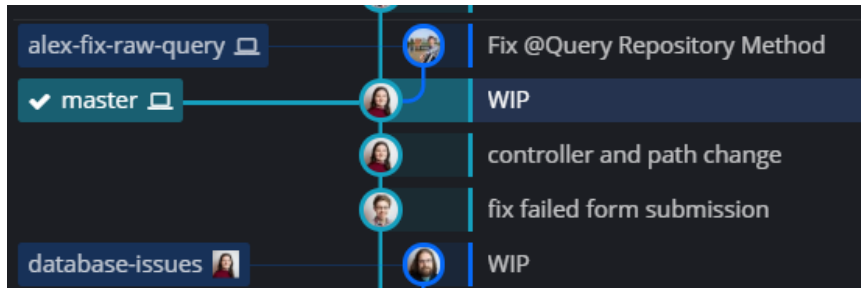
And uploading the key to GitHub as described [here](#)
(Ask a tutor if you want more details)

Refresh the page and now you should be able to see your pushed code. Please show the results to your tutor when we come by.

Resetting

In Git, branches are merely a movable pointer/label on a commit. The label moves when you commit to that branch.

Resetting is a branch operation that moves the label directly:



Resetting (Continued)

The command for resetting is

```
$ git reset [--soft] [--hard] <location (can be branch, commit hash, etc.)>
```

A **soft** reset (--soft) will only move the position of branch label (meaning that git will now notice a lot of staged changes)

Using a **hard** reset (--hard) will change all tracked files to match the new commit along with removing any new files that were added since.

The location is very flexible and you can even specify commits relative to the position of the current branch:

```
$ git reset --hard HEAD~2 # Move the current branch back 2 commits
```


Step 3

Reset

Create a new branch called "reset", checkout this branch and then reset the branch to an earlier commit (try a soft and then a hard reset).

- Can you see the difference between a soft and hard reset?
- You can reset using a relative commit (e.g. HEAD~1) or by finding out the commit id for an existing commit.
 - Try finding the id for an existing commit – your visualizer will offer one way, or try using `git log`. (Hint: “q” will quit the log!)

Revert

Reverting a commit undoes the changes made in that commit.

Checkout the master branch. Next create and checkout a branch called "revert". Finally create a revert commit for the file addition (hint: use `git revert`)

Look at your git graph in a visualizer – has it done what you'd expect?

Step 4

Next, let's create a merge request:

- a) Create a new branch (you can call it something like **pow-operator**)
- b) Change the code from part 1 by adding a new operator, **pow**

```
calc ^ 2 5
```

- c) Afterwards, make a commit and push the new branch to origin
- d) On the repository page, go to Merge Requests and create a **New merge request** (you can also add a comment about the changes you made). For now, **do not merge it**.

Step 5 (Page 1 of 2)

Finally, let's resolve a merge conflict:

- a) Checkout master and create a new branch, this time called **mod-operator**.
- b) Change the code from part 1 by adding a new operator, **mod**.

```
calc % 2 5
```

For the purposes of this exercise, you should have some overlapping work with the **pow-operator** branch (e.g. new work inside the same if-clauses, or adding functions at the end of the files, etc).

- c) Make a commit and push the new branch to origin.

Step 5 (Page 2 of 2)

- d) Merge the **pow-operator** branch (you can also then safely delete the branch).
- e) Create a new merge request, this time for **mod-operator** branch. This time though, you will get some warnings about some existing conflicts.
- f) Go ahead and press **Resolve conflicts** button and resolve the conflicts GitHub couldn't solve on his own. For each file, when you finished, press **Mark as resolved**.
- g) After resolving all the conflicts, commit merge and merge the branch into master.

Rebasing Branches

Rebasing is where a branch's commits are replayed on top of another branch. It can be used as an alternative to merging branches.

This can be useful for simplifying a repository's commit history but does change that history permanently (so is not without risk).

The most common use for this is where you want to replay a feature branch on top of the latest state of the master branch:



Rebasing Branches (Continued)

The command for rebasing is:

```
$ git rebase <target branch> # Assumes checked-out branch is the branch to be rebased  
$ git rebase <target branch> <branch to rebase> # Shorthand to checkout the branch first
```

Interactive rebases provide customisation of how the branch's commits should be replayed (typically an editor will open providing you with instructions on your customisation options):

```
$ git rebase -i <any rebase details>
```

Warning: Like merging, rebasing can lead to merge conflicts!

Step 6

Finally, let's try an interactive rebase:

- a) Create a branch called pre-merge that 2 commits behind master (i.e. before the 2 merge commits that were created in step 4).
- b) Create a new branch from master called ***squashed-merge-commits***.
- c) Interactively rebase ***squashed-merge-commits*** onto pre-merge, combining the 2 merge commits into 1 and changing the commit message to reflect both commits.

If you would like to experiment more with git branching, try
<https://learngitbranching.js.org/>

Python Libraries & Flask

Part 3: Python libraries and the Flask web framework

Recap

Python has a rich ecosystem of libraries and frameworks that have been written to solve common requirements.

Packages can be installed via the pip package manager:

```
$ pip install some-package-name
```

And can then be imported as modules in python scripts:

```
1 from flask import Flask
2
3 app = Flask(__name__)
```

Python also has built in modules that don't need to be installed via pip (e.g. datetime).

Web Applications

Refers to any computer program that contains dynamic logic and deals with user-generated data, which users interact with via the World Wide Web (either through their browser or other client software). This could be

- A website with lots of dynamic content
- A desktop app that interfaces with the web (e.g. most messaging apps)

Web apps typically rely upon a client-server architecture where the logic and content is provided by an application running on a server, but is displayed and runs in a user's web browser (the client).

Flask

Flask is a popular web application framework that provides features such as:

- Running a local server

```
1  from flask import Flask
2
3  app = Flask(__name__)
4
5  @app.route('/hello')
6  def hello_world():
7      return 'Hello World!'
```

```
$ flask run
```

Flask (continued)

- HTTP Request routing

```
1 @app.route('/books')
2 def get_books():
3     # Code to fetch all book entries from the database and return their details.
4
5 @app.route('/books/<id>')
6 def get_book(id):
7     # Code to fetch the book entry with matching id from the database and return its details.
8
9 @app.route('/books', methods=['POST'])
10 def add_book():
11     # Code to create a new book entry in the database.
```

Part 3: Exercise

The Goal

Installing the Flask library and setting up some endpoints covering the use of:

- Serving HTML pages
- Templating
- Submitting and processing forms

Installing a command line library argparse and setting up a command line interface (CLI) to generate data

Step 1:

Install the [Flask](#) python library, as you would any package

```
$ pip install some-package-name
```

Check that you can run a simple Flask app like that on the right, using:

```
$ flask run
```

Hints

- Name your Python file “app.py”
- When ready, visit <http://localhost:5000/hello> in a browser

```
1 from flask import Flask
2
3 app = Flask(__name__)
4
5 @app.route('/hello')
6 def hello_world():
7     return 'Hello World!'
```

Step 2:

Next set up a route that serves the HTML page on the right.

You should serve this under the route `"/films/list"` on your flask web server.

Hints:

- *Place the HTML file in a folder called `templates`*
- *Use [render template](#) from the flask library*
 - *See the example on the next slide if you're not sure how!*

```
<!doctype html>
<html>
  <head>
    <title>All films</title>
  </head>
  <body>
    <ul>
      <li>Flubber</li>
      <li>Jumanji</li>
      <li>Aladdin</li>
    </ul>
  </body>
</html>
```

Flask Templating

```
books = [  
    { 'id': 1, 'title': 'Clean Code', 'authors': 'Robert C. Martin' },  
    { 'id': 2, 'title': 'The DevOps Handbook', 'authors': 'Gene Kim, Jez Humble, Patrick Debois, John Willis' },  
    { 'id': 3, 'title': 'The Phoenix Project', 'authors': 'Gene Kim, Kevin Behr, George Spafford' }  
]
```

```
1 @app.route('/books')  
2 def get_books():  
3     books = get_all_books_from_db() # Some function that returns the list of book entries from the database.  
4     return render_template('book_list.html', books=books)
```

This specifies the
name that will be
available within the
HTML

And this refers to the
variable that exists
locally

Flask Templating (continued)

book_list.html

```
1  <!doctype html>
2  <html>
3    <head>
4      <title>All Books</title>
5    </head>
6    <body>
7      <ul>
8        {% for book in books %}
9          <li>{{ book.title }} - {{ book.authors }}</li>
10         {% endfor %}
11      </ul>
12    </body>
13  </html>
```

If you're new to HTML, remember:

Inside the `<body>` is where we put elements we want to display on the page

`` is an “unordered list”

`` is a “list item” – inside an unordered list, these will appear as bullet points

Step 3:

Next, create a text file containing a list of films and ratings in the following format (known as “csv”):

```
Flubber,3  
Jumanji,5  
Aladdin,4
```

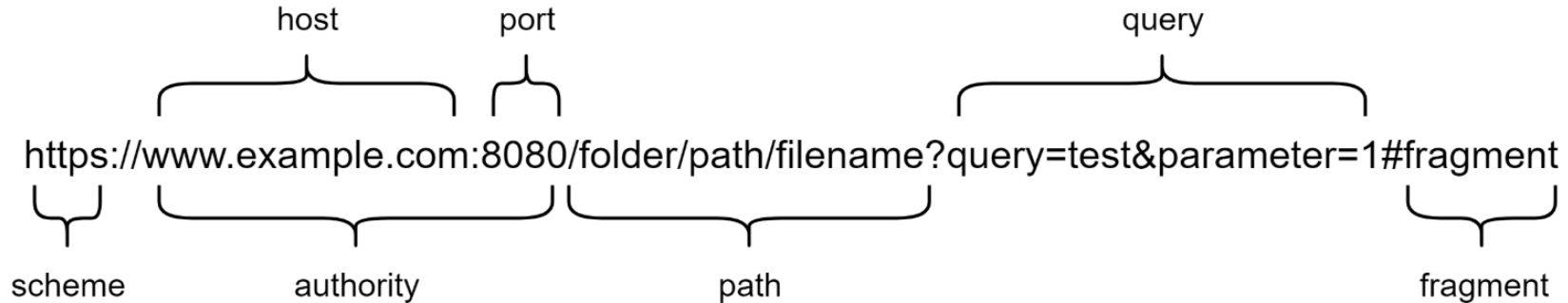
Serve a templated table of films along with their star ratings on the route "films/table”:

```
Film Stars  
Flubber 3  
Jumanji 5  
Aladdin 4
```

Hint: We’ve covered how to do this in earlier slides!

Handling Query Parameters

Recall that query parameters are part of the anatomy of a URL:



They are often used as parameters to HTTP GET requests

Reading Query Parameters In Flask

```
from flask import Flask, request

app = Flask(__name__)

@app.route("/show-message")
def echo():
    name = request.values.get("name", "")
    message = request.values.get("message", "")

    return f"Hey there {name}! You said {message}"
```

If you queried the path

```
/show-message?name=Alex&message=Hello%20Everyone!
```

The response would be

```
Hey there Alex! You said Hello Everyone!
```

Step 4:

Now add support for filtering this list by star rating, e.g. "films/table?stars=3" should return just:

Film Stars
Flubber 3

You can show the tutor your progress to confirm that everything is working as expected.

Step 5:

Use the built in [argparse](#) python library and use it generate a command line app for submitting film reviews. Running this command should add “flubber,3” as a new line in your csv file.

```
python cli_app.py --film-name=flubber --stars=3
```

The app should support submitting multiple reviews (by repeatedly running the app) to build up a list of many films.

Note: If you're having trouble getting argparse to work, feel free to use python's inbuilt “input” function instead to read values from the command line. There's also a tutorial for using the library [here](#).

Template Includes and Extends

The templating library for Flask, Jinja, has various mechanisms for composing HTML templates together.

The simplest is a direct inclusion of one template within another. For example you could have:

```
{% include 'header.html' %}  
    Body  
{% include 'footer.html' %}
```

Which includes 2 external templates corresponding to the head and footer respectively.

Please note that included templates can also use the same templated parameters as the master template.

Template Includes and Extends (continued)

It is also possible for one template to “inherit” from another template using the extends keyword:

Base Template

```
<!DOCTYPE html>
<html lang="en">
<head>
  <link rel="stylesheet" href="style.css" />
  <title>{% block title %}Default Title{% endblock %} - My Webpage</title>
</head>
<body>
  <div id="content">{% block content %}{% endblock %}</div>
</body>
</html>
```

Child Template

```
{% extends "base.html" %}
{% block title %}Index{% endblock %}
{% block content %}
  <h1>Index</h1>
  <p class="important">
    Welcome to my awesome homepage.
  </p>
{% endblock %}
```

This can be useful for sharing a common page structure across a website.

Step 6:

Finally build an HTTP form served on the route `"/films/submit"` that fulfils the function of step 5.

If you've got this working feel free to experiment with the include/excludes templating structures introduced today or play around with the features listed on the [Jinja Template Designer Documentation](#).

How did it go?

Thank You!



Please complete the feedback survey by
scanning the QR code or clicking [here](#)