

Configuring programs and using APIs

Data Sciences Institute
University of Toronto

Simeon Wong

Introductions

Meet your instructor



Simeon Wong

4th year PhD @ Institute of Biomedical Engineering
Research Analyst @ Hospital for Sick Children

- I read minds!
(of children with epilepsy using implanted electrodes)
- MRI, CT image processing
- EEG, MEG signals processing
- Neuromodulation platforms for research

Meet your TA

Tong Su

- 4th year undergrad @ University of Toronto
- Computer Science Specialist and Statistics Major
- Full-stack developer at Northbridge (Co-op)
- Research Interest: Computer Vision, Natural Language Processing, and Machine Learning

Asking questions

- Zoom chat during class
 - Feel free to post and answer questions at any time
 - I will pause for questions occasionally, and review questions from the chat
- Pre- / Post-class office hours with Tong
- Email
 - simeonm.wong@mail.utoronto.ca
 - tong.su@mail.utoronto.ca

Daily scrum

Write in the chat:

1. How you'll use one thing you've learned so far
2. One thing you're struggling with / wish the course addressed
3. What you're looking forward to in this course

Course overview

1. Configuration files & Environment variables
2. Using and writing Application Programming Interfaces (APIs)
3. Handling errors
4. Testing software
5. Building Python packages
6. Working in software teams using GitHub features

Course overview

- 1. Configuration files & Environment variables**
- 2. Using and writing Application Programming Interfaces (APIs)**
3. Handling errors
4. Testing software
5. Building Python packages
6. Working in software teams using GitHub features

Assessments

- Outcomes-based learning
 - > 75% of course outcomes
- Assessed with both in-class exercises and summative project
- Summative project:
 - The next 4 classes build on each other
 - We will work on the summative project step-by-step through each class

Today's learning outcomes

- I can interpret and write simple YAML files
- I can load YAML files into Python
- I can read command line arguments from Python
- Given documentation, I can use a Python or HTTP API

Configuring programs

Why configure?

- Programs are only useful if they can be **controlled**.
- Work inside programs is **reproducible** only if controls are **explicit** (documented and understandable) and **shareable**.
- Programs may need to work differently in different contexts:
 - Operating systems
 - Data directories
 - Localization: language, region settings
 - Analytical methods or outputs (e.g. linear vs quadratic regression)

Configuring programs

Data sources

- Data that controls how a program functions can come from multiple sources:
 - Context
 - Current operating system
 - System language
 - Files
 - Configuration files or job description files
 - Dynamic sources
 - Network server, Secrets manager, Azure/AWS/GCP configuration server, etc...

Configuring programs

Layers of configurations

1. Default program configuration
2. System-wide configuration
 - General settings that impact all users (and multiple programs)
3. User-specific configuration
 - Personal preferences
4. Job-specific configuration
 - Run-specific information
5. Command-line options
 - Frequently changed, or on-the-fly configuration

Configuring programs → Layers of configurations

Default program configuration

- Sensible defaults included with the program
 - Designed to be generally useful for most people who run the program
 - e.g. matplotlib default colour is blue
- Consider different types of users
- Be careful of unexpected behaviour if left unconfigured
 - e.g. Program that saves data to `financials.xlsx` by default
 - Might unintentionally overwrite users' files
 - Some defaults should not be set to force users to specify

Configuring programs → Layers of configurations

System-wide configuration file

Purpose: To provide default settings applicable to all users of a system.

Impact: Changes affect every user and application on the system.

Modification and Risks: Editing system-wide files requires caution as incorrect settings can impact system stability.

e.g: Modifying a system-wide configuration file in an operating system to change the default network timeout settings. This change will affect all network-related operations for all users, even those using other programs.

Configuring programs → Layers of configurations

User-Specific Configuration

Scope: Settings that apply only to a single user's environment.

Flexibility: Allows personalization without affecting other users.

Storage: Typically stored in user's home directory or specified user profile sections.

e.g: A user creates a configuration file in their document editor to set a default font size and page layout, different from the system-wide defaults.

Configuring programs → Layers of configurations

Job-Specific Configuration

Application: Used for settings that apply to a specific task or project.

Priority: Overrides system and user-specific settings for the job's duration.

Creation and Use: Crafted for individual projects or tasks, often located within the project directory.

e.g: Setting up a configuration file in a data analysis project to specify data sources and output formats unique to that project.

Configuring programs → Layers of configurations

Command-line options

Use: For temporary adjustments or one-off changes.

Flexibility: Allows quick, on-the-fly changes without altering permanent configurations.

Best Practices: Use for frequent or minor changes; avoid for complex configurations.

e.g: Running a file compression tool with command-line options to set a high compression level for a specific large file, overriding the default compression setting.

Configuring programs

Layers of configurations

1. Default program configuration
2. System-wide configuration
 - General settings that impact all users (and multiple programs)
3. User-specific configuration
 - Personal preferences
4. Job-specific configuration
 - Run-specific information
5. Command-line options
 - Frequently changed, or on-the-fly configuration

Configuring programs

Overlay/override system

- The same option is valid in multiple configuration layers
- Options read from more specific layers override settings configured in more general ones

e.g: Plot colours for an analysis program

- Program's default colour is blue (matplotlib default)
- Your organization's brand colour is green (system-wide)
- For this specific analysis, need to use orange (job-specific)

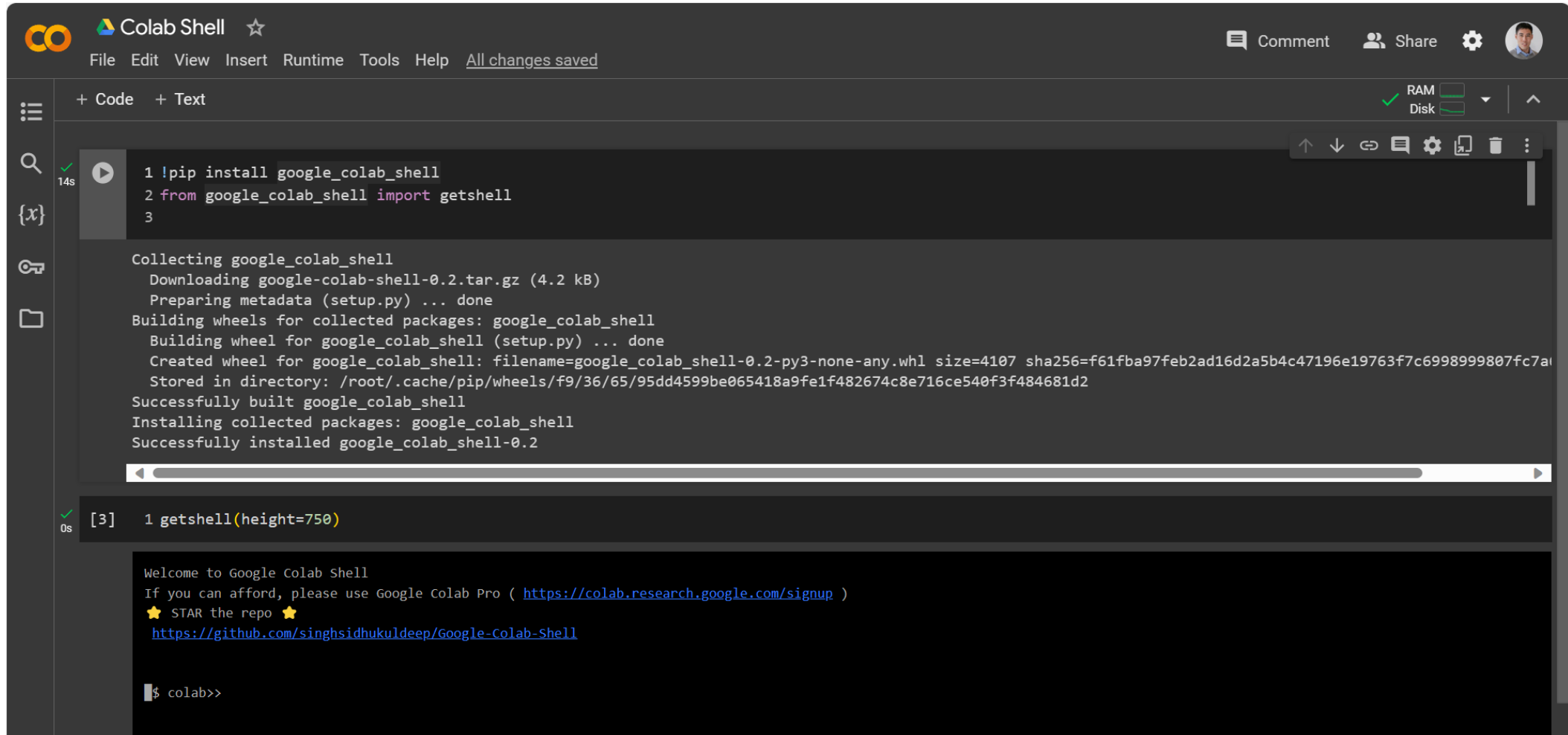
What questions do we have?

Demo: Google Colab Terminal

- UofT DSI uses Google Colab for Python
- Parts of this course will require using Git and Python from the command line (instead of the notebook interface)
- **If you have a Unix shell, Git, and Python installed locally:**
feel free to use that
- **If you are not sure or you do not:**
we can access a basic shell on Google Colab

Course logistics

Demo: Google Colab Terminal



The screenshot shows a Google Colab terminal interface. At the top, there's a header with the Colab logo, 'Colab Shell', and a star icon. Below this is a menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', 'Help', and a link 'All changes saved'. On the right side of the header, there are icons for 'Comment', 'Share', a settings gear, and a user profile. Below the header, there's a toolbar with '+ Code' and '+ Text' buttons. The main area is divided into two sections. The top section shows a code cell with three lines of Python code:

```
1 !pip install google_colab_shell
2 from google_colab_shell import getshell
3
```

 The bottom section shows the output of the code cell, which includes the installation process for 'google_colab_shell'. The output text is:

```
Collecting google_colab_shell
  Downloading google-colab-shell-0.2.tar.gz (4.2 kB)
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: google_colab_shell
  Building wheel for google_colab_shell (setup.py) ... done
  Created wheel for google_colab_shell: filename=google_colab_shell-0.2-py3-none-any.whl size=4107 sha256=f61fba97feb2ad16d2a5b4c47196e19763f7c6998999807fc7a1
  Stored in directory: /root/.cache/pip/wheels/f9/36/65/95dd4599be065418a9fe1f482674c8e716ce540f3f484681d2
Successfully built google_colab_shell
Installing collected packages: google_colab_shell
Successfully installed google_colab_shell-0.2
```

 Below this, there's a prompt '[3] 1 getshell(height=750)' and the output of the command, which is a welcome message:

```
Welcome to Google Colab Shell
If you can afford, please use Google Colab Pro ( https://colab.research.google.com/signup )
★ STAR the repo ★
https://github.com/singhsidhukuldeep/Google-Colab-Shell

$ colab>>
```

Configuring programs

Command-line arguments in Python

- The **argparse** library built into Python is used to interpret command-line arguments
- Common types of command-line arguments
 - Positional argument (e.g. `mv oldpath newpath`)
 - Flags (true/false values) (e.g. `ls -l`)
 - Options with values (e.g. `head -n 5`)

\$> Interactive live coding

1. Create a Python script in Google Colab (or on your local computer)
2. Make it print “Hello World!”
3. Referring to the **argparse** documentation:
 1. Initialize **argparse**
 2. Write a short description
 3. Add a **positional argument** for greeting target
 4. Add an **option** for number of repeats
 5. Add a **flag** for saying goodbye

What questions do we have?

Configuration file formats

Popular Formats:

- INI – older format for simple, structured data
 - Commonly found in compiled Windows / Linux programs
- JSON – older standard, flexible, can be difficult to type by hand
- YAML – new standard, readable and flexible

Due to its flexibility and readability, especially for complex configurations, **YAML is recommended**.

Configuring programs

The YAML format

```
key: value
```

```
dict:
```

```
  key1: value1
```

```
  key2: value2
```

```
list:
```

```
  - value1
```

```
  - value2
```

```
# comment
```

“YAML is Another Markup Language”

Online tools for browsing YAML:

<https://codebeautify.org/yaml-parser-online>

Configuring programs

\$> Interactive live coding

1. Create a YAML file in Google Colab
2. Write Python code to load and consolidate YAML files
- 3. Exercise:** Write your own configuration files
 - For an analysis where a dataset is loaded and two columns from the dataset are plotted on the x and y axes in your favourite colour
 - Variables: `dataset_url` (str), `cols_to_plot` (dict with x and y str), `color` (str)
 - Which configuration levels do these variables go into?

What questions do we have?

Lesson overview

Today's overview

1. Configuration files & Environment variables
2. Using and writing Application Programming Interfaces (APIs)

Application Programming Interfaces

What is an API?

- API stands for Application Programming Interface
- Allows software programs to communicate with each other
- Provides structured way to communicate between applications and devices (expose data and functionality)
- Allows other developers to access and integrate with an application without needing to understand complex implementation details

Application Programming Interfaces

Programs using programs

- You've already used APIs!
- Python APIs: matplotlib, numpy, pandas
- Web APIs: City of Toronto open data API

Public APIs vs Private APIs

- Public vs private refers to access, visibility, and documentation

Public APIs

- Available openly for any developer to use
- Well-documented and robustly coded to account for different (and untrusted) requests and input from the public
- Public Web APIs: Just need to sign up and get an API key to access
- e.g. GitHub, Spotify, YouTube

Public APIs vs Private APIs

Private APIs

- Access is restricted to internal apps or trusted external partners and requires authorization
- May be coded/documented for very specific use cases
- For interacting with internal data and functionality safely
- e.g.: APIs for internal tooling, bank partnerships

Application Programming Interfaces

Public APIs vs Private APIs

- When writing APIs for public use:
 - Must validate inputs strictly
 - Defend against coding mistakes
 - Defend against malicious users (e.g. access to unauthorized data, system compromise)
 - Must document extensively
- Tradeoff between additional utility and engineering-hours

What questions do we have?

Application Programming Interfaces

The RESTful Web API

- The RESTful Web APIs are a quasi-standard method of performing actions using or exchanging data with web-connected services
 - e.g. Retrieve list of repositories from GitHub
 - e.g. Using GPT-4 to process datasets automatically
 - e.g. Starting and stopping a container hosted on Microsoft Azure
- REST: “Representational State Transfer”
 - Uses HTTP requests: GET, POST, (PUT), (DELETE)
 - Generally, returns data in machine-readable formats like JSON

Application Programming Interfaces

The RESTful Web API

- Uniform interface
 - Every entity (piece of data) is generally retrieved from the same URI
- Client-server decoupling
 - The web interface is assumed to be the only link between the client and the server
 - Data isn't getting passed through a separate channel (e.g. file on hard drive)
- Statelessness
 - All required information is included in the request
 - Identity is established on every request (usually via a secret token)

Application Programming Interfaces

The RESTful Web API

Client

GET /user
Authorization Bearer zn4 ... 2l3

*“Get details about myself.
I am identified by ...”*



GitHub API (https://api.github.com)

```
{ "login": "octocat", "id": 1,  
  "node_id": "MDQ6VXNlcjE=",  
  "avatar_url":  
    "https://github.com/images/error/octocat_happy.gif",  
  "gravatar_id": "",  
  "url":  
    "https://api.github.com/users/octocat",  
  "html_url":  
    "https://github.com/octocat", ... }
```

*“Here are your details...
.. beeeepboop ...”*



Application Programming Interfaces

The RESTful Web API

Client

DELETE /repos/owner/thisrepo
Authorization Bearer zn4 ... 2l3

*"Delete this repository!
I am identified by ... "*



GitHub API

(<https://api.github.com>)



Status: 204

"Done!" 🌟

Application Programming Interfaces

API keys

- Unique string that acts like a password obtained by registering as a developer
 - Usually time-limited (hours to months)
- Identifies the client: Usage tracking and access limits
- API keys are usually sent in the request header
 - e.g. `Authorization: Bearer 23748237842823442`
- Keep your key secret - don't share or expose!
 - Store using secrets manager or protected configuration file (.gitignore is your friend!)

Application Programming Interfaces

Web API responses

- Consists of an HTTP response code + body
- Response codes:
 - 2xx = success (e.g. 200)
 - 4xx = error with the request (e.g. 404 URI not found, 400 bad request)
 - 5xx = error with the server (e.g. 500 internal server error)
- The body is generally in JSON format (rarely, but sometimes in XML)

What questions do we have?

Making Web API requests in Python

- Use the **requests** library to communicate with the API
- Use the **json** library to parse JSON responses
- Sometimes companies release Python libraries that make it easier to use their APIs from Python
 - Simplify authentication, request validation, response parsing, etc...

Application Programming Interfaces

\$> Interactive live coding

1. Generate a GitHub Personal Access Token (API key)

Settings > Developer Settings > Personal access tokens

2. Add token to Colab secrets manager



This is a type of configuration!

3. Refer to GitHub API Documentation
4. Use **requests** to retrieve own user details from GitHub Web API

Application Programming Interfaces

\$> Interactive live coding

```
1 import requests
2 import json
3 from pprint import pprint
4
5 from google.colab import userdata
6
7 token = userdata.get('ghtoken')
8
9 response = requests.get(url='https://api.github.com/user',
10                          headers={'Authorization': 'Bearer ' + token})
11
12 # print raw response
13 print(response.status_code)
14 print(response.text)
15
16 # parse json
17 response_json = json.loads(response.text)
18 pprint(response_json)
19
20 # print some values
21 print('Username: ' + response_json['login'])
22 print('Name: ' + response_json['name'])
23
200
{"login": "dtxe", "id": 7825879, "node_id": "MDQ6VXNlcjc4MjU4Nzk=", "avatar_url": "https://avatars.githubusercontent.com/u/7825879?v=4", "gravatar":
{"avatar_url": "https://avatars.githubusercontent.com/u/7825879?v=4",
'bio': 'PhD student in Biomedical Engineering at SickKids and the University '
      'of Toronto.\r\n'
      'Building responsive neuromodulation strategies for children with '
      'epilepsy.',
'blog': 'https://simeonwong.com',
'company': 'Hospital for Sick Children',
'created_at': '2014-06-07T17:32:08Z',
'email': None,
'events_url': 'https://api.github.com/users/dtxe/events{/privacy}'.,
'followers': 7,
'followers_url': 'https://api.github.com/users/dtxe/followers',
'following': 1,
'following_url': 'https://api.github.com/users/dtxe/following{/other_user}'.,
'gists_url': 'https://api.github.com/users/dtxe/gists{/gist id}'.,
'gravatar_id': '',
'hireable': None,
'html_url': 'https://github.com/dtxe',
'id': 7825879,
'location': 'Toronto, ON',
'login': 'dtxe',
'name': 'Simeon Wong',
```

What questions do we have?

Exercise: Explore the GitHub API

- Using the GitHub API documentation:
 - choose an interesting endpoint that returns data
 - write a Python request to retrieve data from that endpoint
 - *bonus*: visualize or analyze it in some way
- Ideas:
 - List the top 20 most starred repositories using /search/repositories
 - List the top 20 most followed users using /search/users
 - Hint: try using q=stars:>1 or q=followers:>1

What questions do we have?

Today's learning outcomes

- I can interpret and write simple YAML files
- I can load YAML files into Python
- I can read command line arguments from Python
- Given documentation, I can use a Python or HTTP API

Homework

- Create a personal GitHub repository for today's work
 - We suggest making it public
 - Otherwise, private with dtxe and sue-tong as collaborators
- Commit your Hello World and your GitHub API code
 - Write a suitable commit message
 - Write a simple README.md file
- Push your commit and submit your work here:
<https://uoft.me/dsi-1-bs-a1>

References

- Research Software Engineering with Python by Damien Irving, Kate Hertweck, Luke Johnston, Joel Ostblom, Charlotte Wickham, and Greg Wilson (<https://merely-useful.tech/py-rse/config.html>)