CS162 Solo Project Weekly Report: Week 03

CS162, Intro. to Computer Science II
Semester 2, AY2023/24

Solo Project

Full Name	Student ID
NGUYEN Hoang The Kiet	23125023

Lecturer: DINH Ba Tien (PhD)
TAs: HO Tuan Thanh (MSc), NGUYEN Le Hoang Dung (MSc)

Ho Chi Minh City, March 31, 2024

Tasks by Week

Week 03

1. (Section 1.1) Create a *private* GitHub repository
2. (Section 1.2) Clone a repository from GitHub
3. (Section 1.3) Create new files and folders in GitHub
4. (Section 1.4) `.gitignore` 101
5. (Section 1.5) Basic Git commands `(add, commit, push, pull)`
6. (Section 2.1) Python project with multiple files
7. (Section 2.2) Importing external files
8. (Section 2.3) Python classes, properties and methods

Contents

List of Figures

List of Listings

Chapter 1

GitHub 101

ⓐ | The necessity of a Version Control System (VCS)

When a project enlarges and/or is contributed by a large team of developers, many problems arise that differ from a personal, small-scale project:

- **Non-linear development**. Developers may start from different versions of the code, from which they would grow the project, resulting in a Directed Acyclic Graph (DAG) based development.
- **Distribution**. Developers may work on different aspects of the project, thus there is a need for collaboration platform.
- **Failure management**. When the current version fails, the project may need to roll back to previous versions.

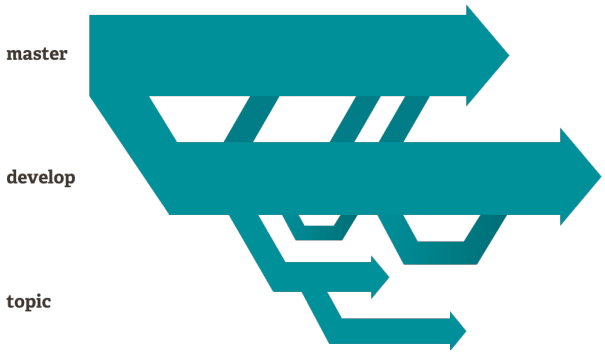


Figure 1.1: Github: Large-scale project version control.
Source: https://git-scm.com/

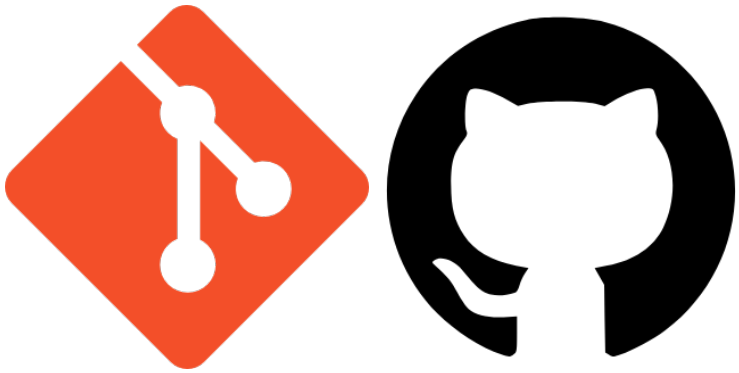


Figure 1.2: Github: The Git logo resembles a version control diagram of a project. The GitHub logo resembles... a cat. Turns out, that cat is no normal cat, it is the Octocat - GitHub's mascot.

ⓑ | Git vs. GitHub

Git

Git is a Distributed Version Control System (DVCS), a local tool used for tracking code changes in a project. Git supports:

- **Snapshotting**. Creating snapshots of the project at different timestamps.
- **Monitoring**. Tracking changes made by different developers.
- **Revertability**. Rolling back to previous versions.

Git exists in both the form of Graphical User Interface (GUI) or Command Line Interface (CLI). *This report will mostly use the command-line version of Git.*

To install Git, download an installer at https://git-scm.com/, or install via packages like `apt`, `yarn`, `chocolatey`, etc.

To confirm that Git is installed, enter `git version` in the Terminal.

```
$ git version
git version 2.33.0.windows.2
```

Listing 1.1: Github: Check if `git` is installed. Display the version of Git.

GitHub

GitHub is a cloud-based service that provides functionalities built on top of Git.

In short, GitHub acts like a cloud storage medium which also supports Git-style version control. To make full use of GitHub, it is recommended to have Git installed.

© | Terminologies

This subsection is based mostly on the official Github's Repositories documentation[1].

repository (Repo)

A repository is the most basic element of GitHub. It's a place where you can store your code, your files, and each file's revision history. Repositories can have multiple collaborators and can be either public or private. A repo can be either *public* or *private*.

clone

To clone is to download a full copy of a repository's data from GitHub.com, including all versions of every file and folder.

commit

To commit is to create a *snapshot* of the code.

branch

A branch is a parallel version of the code that is contained within the repository, but does not affect the primary or `main` branch.

fork

A fork is a new repository that shares code and visibility settings with the original "upstream" repository.

pull

A pull is a request to merge changes from one branch into another.

1.1 | Create a *private* GitHub repository

- **Step 1.** On the top-right corner, click on the + *Create new...* button to reveal the menu. Then click on *New repository*.

  Alternatively, go to https://github.com/new.

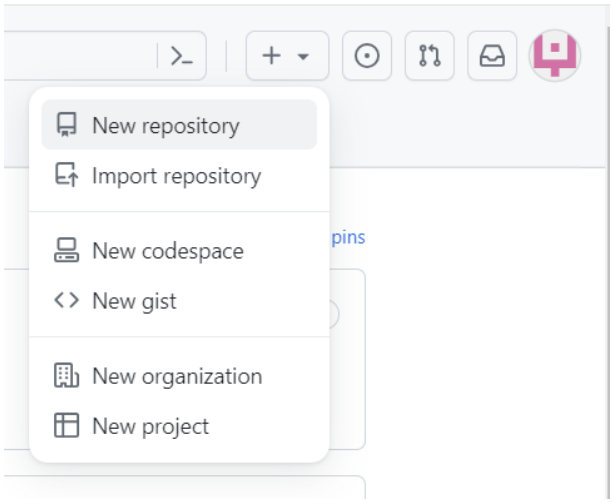


Figure 1.3: Github: Creating a new repository

- **Step 2.** Fill the information into the fields. An example is given in Figure 1.4 Some commonly used fields are
  - **Owner**. By default it is set to the creator.
  - **Repository name**. Must be at most 100 code points (a code point is either an alphanumeric character, a hyphen `'-'`, an underscore `'_'` or a period `'.'`)
  - **Description** *(Optional)*. A brief description of the repository.
  - **Public/Private**. Choose the visibility of the repo. In this specific example, choose *Private*.
  - **Initialise this repository with**. Create default files (`README.md` and `.gitignore`) and select licensing options. For simplicity, it can be left blank until later.

The result should look like in Figure 1.5.

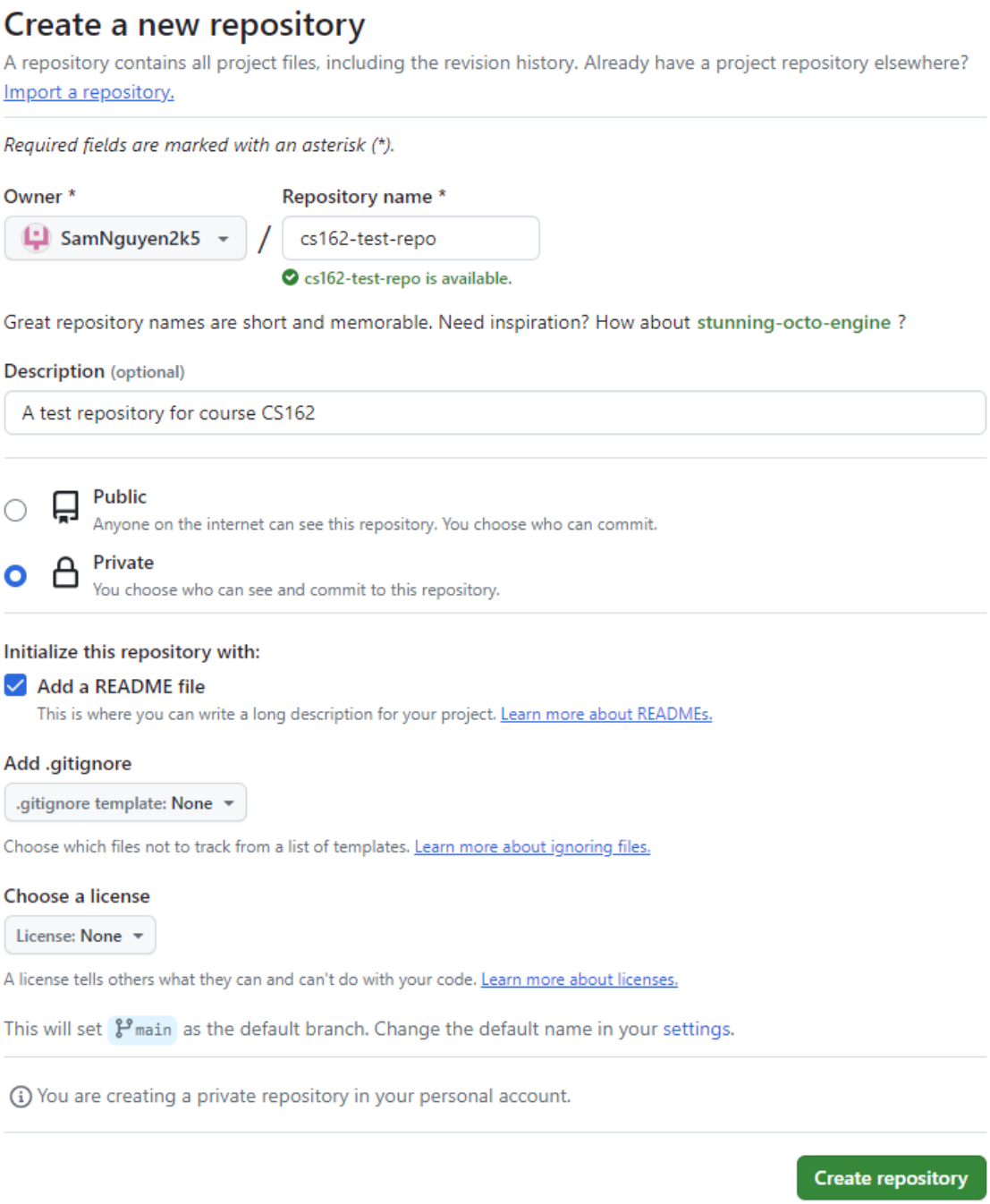


Figure 1.4: Github: Repository information field.

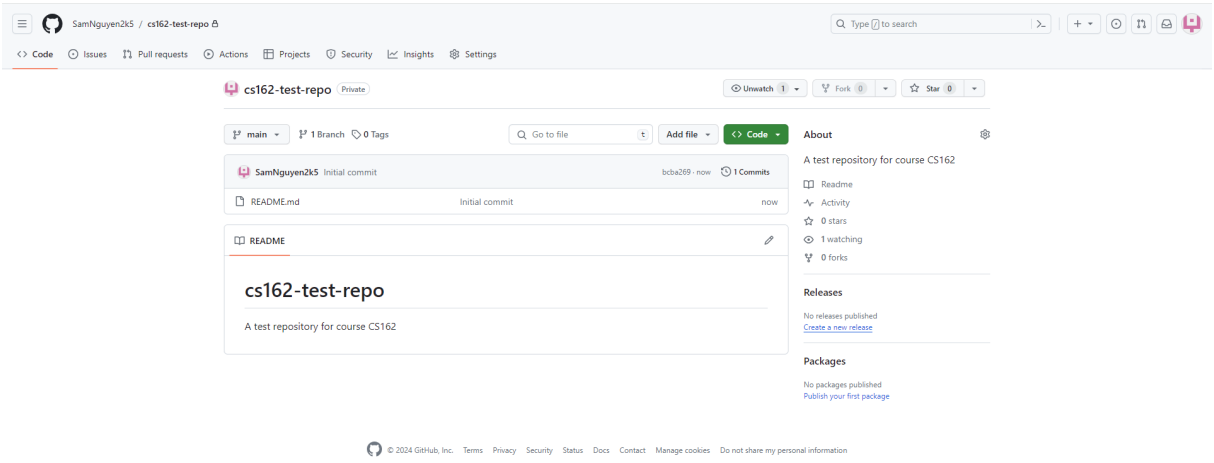
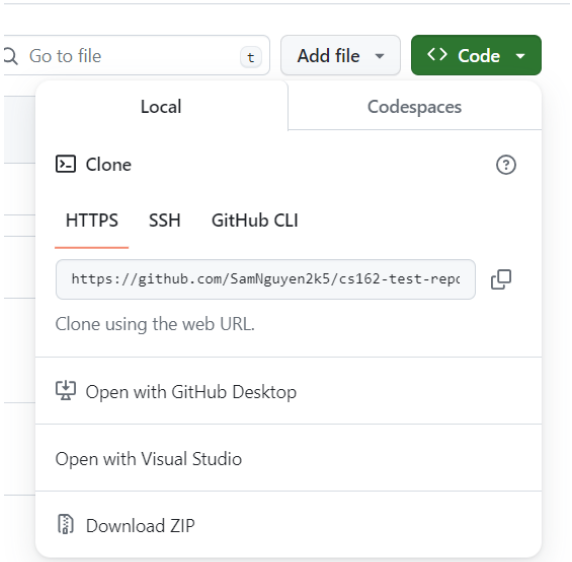


Figure 1.5: Github: Repository successfully created.

1.2 | Clone a repository from GitHub

- **Step 1**. At the main page of the repository, click on 



- **Step 2**. Copy the `.git` URL. *(This URL is for cloning using HTTPS)*.
- **Step 3**. In the Terminal, open the folder into which the repository is to be cloned.
- **Step 4**. Use the `git clone <link-to-repository>` command.

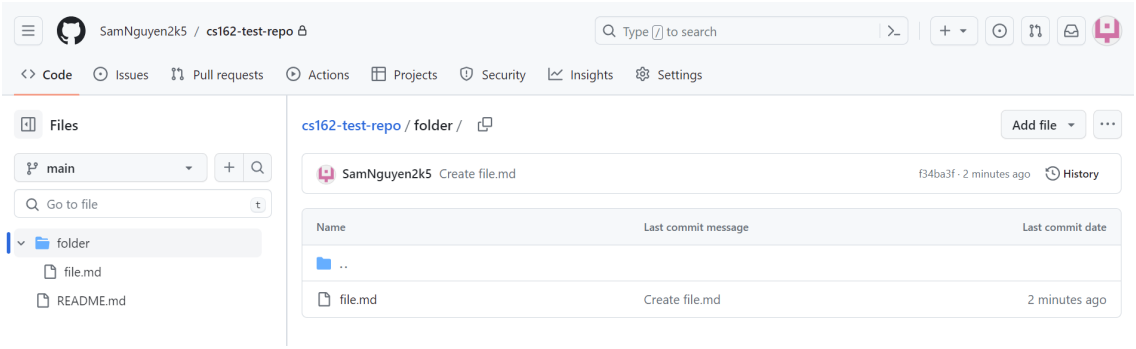
1.3 | Create new files and folders in GitHub

```
$ git clone https://github.com/SamNguyen2k5/cs162-test-repo.git
Cloning into 'cs162-test-repo'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
```

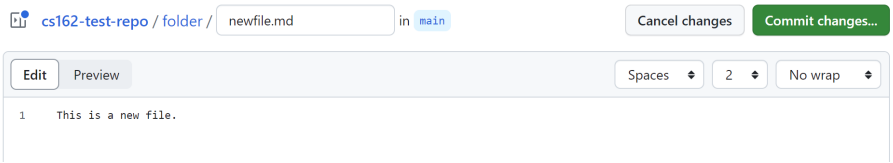
Listing 1.1: Github: Link to `.git` for repository cloning.

(a) | Create files from scratch

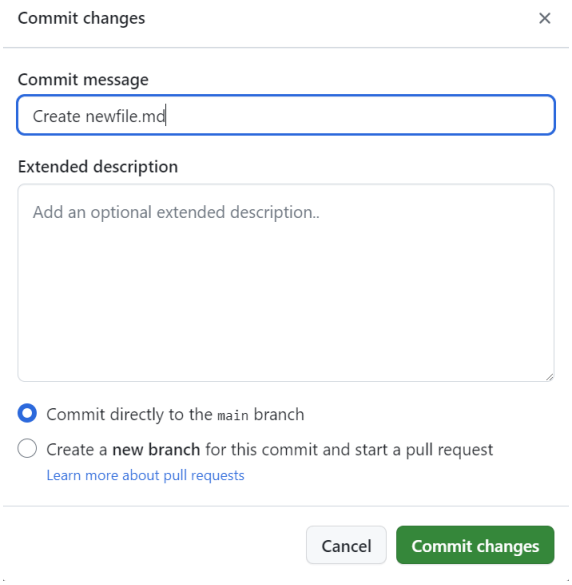
- **Step 1**. Navigate to the folder to which files/folders are added.
- **Step 2**. Select the Add file menu, then click *Create new file*, or select + in the tree view on the left.



- **Step 3**. Set the filename. Subfolders can be created by adding a slash `('/')`.
- **Step 4**. Enter the content of the file.

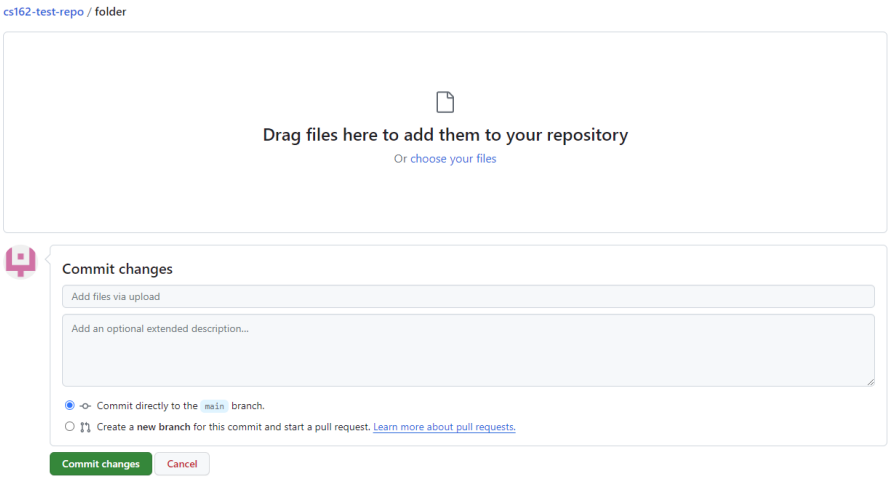


- **Step 5**. Select Commit changes. Write a brief *Commit message*, then choose which branch to commit. The new file can either be
  - Commited directly to the `main` branch.
  - Commited to a new branch, and a pull request is made.



(b) | Upload local files

- **Step 1**. Navigate to the folder to which files/folders are added.
- **Step 2**. Select the Add file menu, then click *Upload files*.
- **Step 3**. Drag files or folders to upload. Then write *commit message* and choose a commit branch, similar to above.



1.4 | .gitignore 101

`.gitignore` is a Git file that lists out all files **to be passed** by Git when commited and pushed to the remote repository. These files usually:

- are unimportant for development, but require substantial amount of repository storage (`.vscode` folder, compiled executable files, etc.)
- contain sensitive when publicised (password database, etc.)

`.gitignore` is a text file in which each line is a folder path or a file to be ignored by Git.

```
## This is a comment
#### A comment starts with a hash character ('#')

# Ignore a folder
.vscode

# Ignore a file
main.aux

# Ignore a file with wildcard
*.log
debug.log*
```

Listing 1.1: Github: Example of `.gitignore`

1.5 | Basic Git commands (add, commit, push, pull)

ⓐ | Syntax

- **`git add`**. Add files/folders to the staging environment (start tracking files/folders).

```
git add <file>
```

- **`git commit`**. Snapshot a change to the **local repository**.

```
git commit -m "Commit message"
```

- **`git push`**. Transfer local commits to the remote repository.

```
git push <remote_name> [<local-branch_name>:]<remote-branch_name>
```

- **`git pull`**. Update local repository from the remote repository.

```
git pull <remote> <branch>
```

ⓑ | Example

`git add`

```
$ tree /f
Folder PATH listing for volume Windows-SSD
Volume serial number is 62AA-618C
C:.
|   README.md
|
+---folder
        file.md

$ vim added_file.md

$ type added_file.md
This file is added by
```
    git add added_file.md
```

$ git add added_file.md

$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   added_file.md
```

Listing 1.1: Github: Example of `git add`

`git commit` and `git push` to the current working branch

```
$ git commit -m "Commit added_file.md"
[main 444661f] Commit added_file.md
 1 file changed, 4 insertions(+)
 create mode 100644 added_file.md

$ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 365 bytes | 365.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/SamNguyen2k5/cs162-test-repo.git
   f34ba3f..444661f  main -> main

nothing to commit, working tree clean
```

Listing 1.2: Github: Example of `git commit` and `git push` on current `main` branch

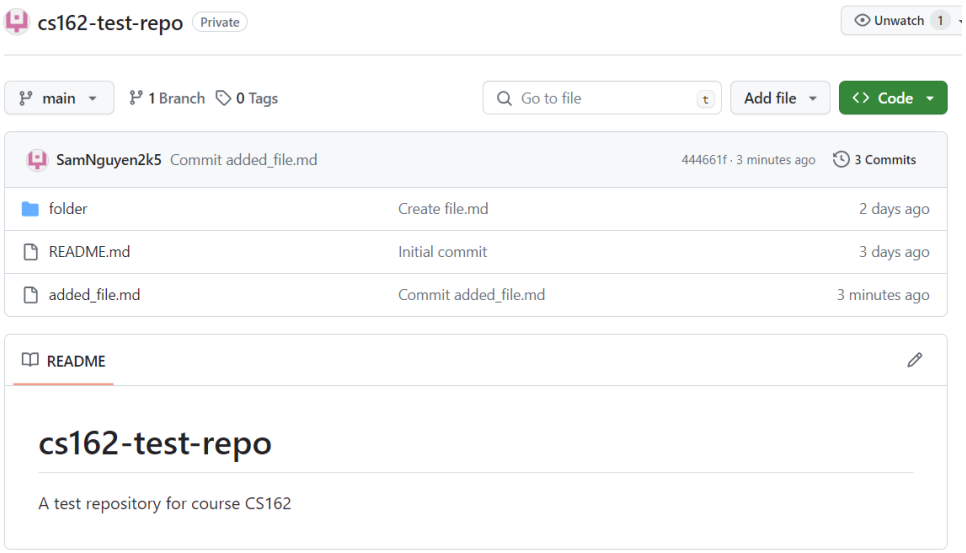


Figure 1.6: Github: After `git push`, `added_file.md` is added into the `main` branch.

`git commit` and `git push` to another working branch

```
$ git commit -a -m "Commit to new branch"
$ git checkout -b other
Switched to a new branch 'other'

$ git push origin other
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 403 bytes | 403.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'other' on GitHub by visiting:
remote:    https://github.com/SamNguyen2k5/cs162-test-repo/pull/new/other
remote:
To https://github.com/SamNguyen2k5/cs162-test-repo.git
 * [new branch]      other -> other
```

Listing 1.3: Github: Example of `git commit` and `git push` on another branch `other`

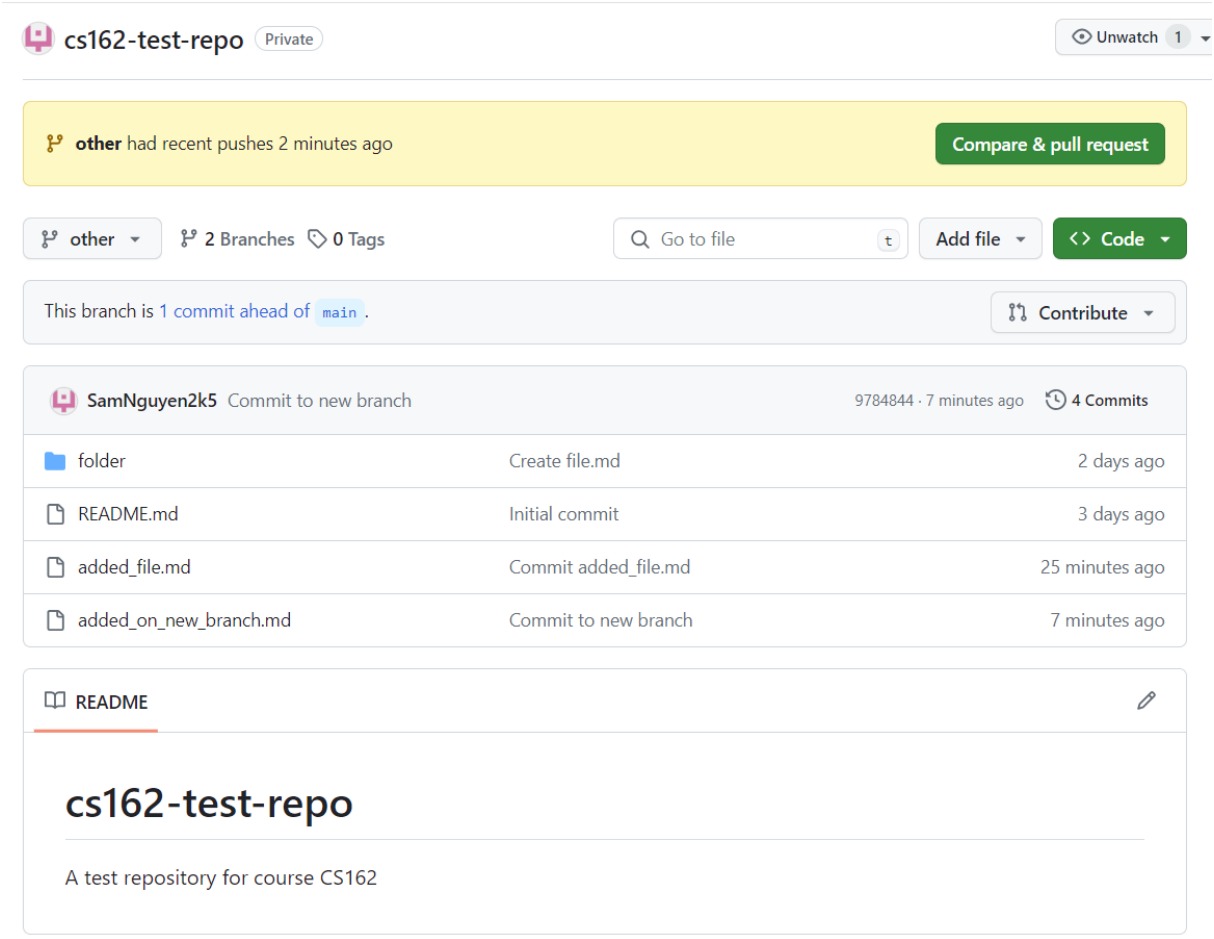


Figure 1.7: Github: Branch (`other`) created with `added_on_new_branch.md` added.

```
git pull
```

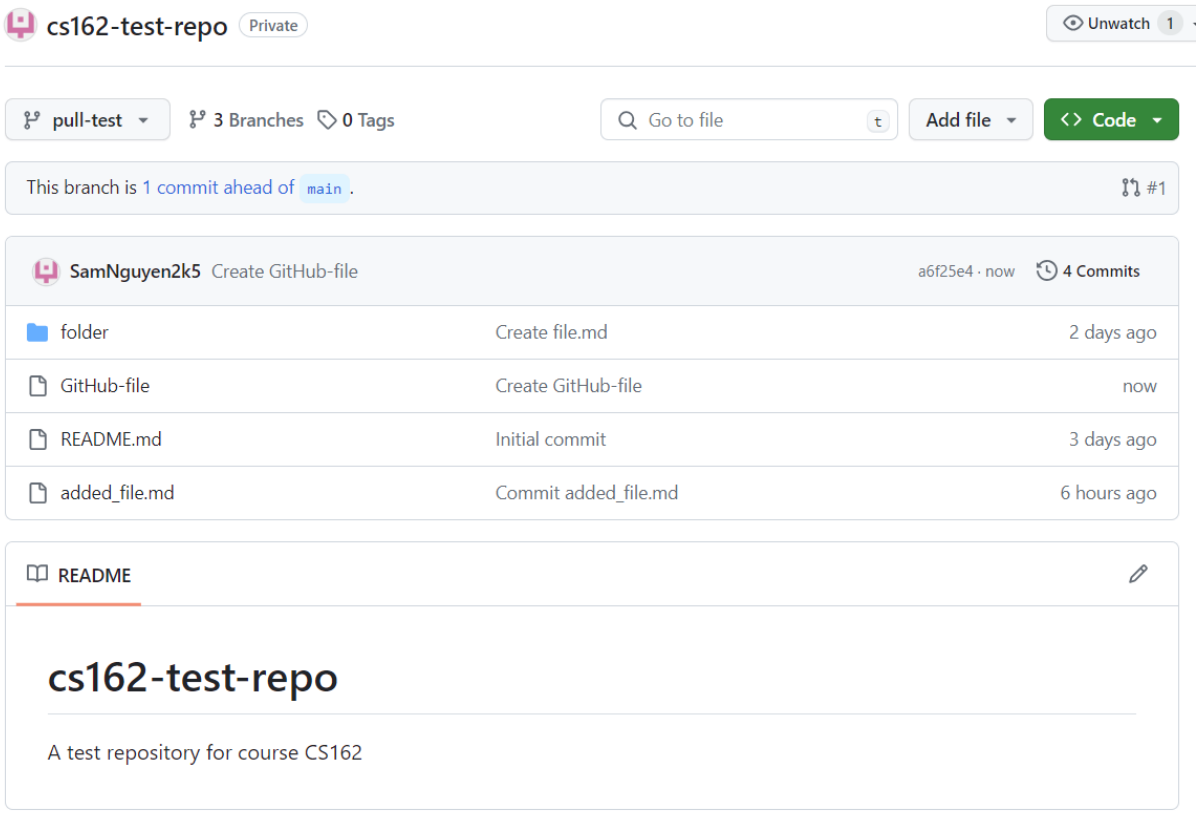


Figure 1.8: Github: A file is created on the GitHub website. The current local repository has no knowledge of this newly created file.

```
$ git pull
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 1.03 KiB | 176.00 KiB/s, done.
From https://github.com/SamNguyen2k5/cs162-test-repo
 * [new branch]      pull-test  -> origin/pull-test
There is no tracking information for the current branch.
Please specify which branch you want to merge with.
See git-pull(1) for details.

    git pull <remote> <branch>

If you wish to set tracking information for this branch you can do so
with:

    git branch --set-upstream-to=origin/<branch> other

$ tree /f
Folder PATH listing for volume Windows-SSD
Volume serial number is 62AA-618C
C:.
   added_file.md
   added_on_new_branch.md
   GitHub-file
   README.md

folder
        file.md
```

Listing 1.4: Github: After `git pull`-ing, `GitHub-file` has been recognised.

Chapter 2

Python 101

With regards to Artificial Intellgence and Machine Learning, Python is a great choice to start, even for researchers with little coding background. Python offers:

- **Easy syntax**. Pythonic syntax is close to human readable text
- **Extensive AI/ML library support**. `numpy`, `pandas`, `scikit-learn`, etc. are examples of beginner-friendly frameworks for Data Analysis and the Basics of Machine Learning. For professionals, many libraries are backed by large tech companies, such as Google (`Tensorflow`, `Keras`), Facebook (`Pytorch`), etc.
- **Platform flexibility**. Python performs uniformly on various operating systems, unlike C/C++ which is platform dependent, i.e. may run differently in different machines.
- **Scalability**. Easy for prototypes, easy to scale.

Figure 2.1: Python: Extensive AI/ML libraries. *Source: Internet*

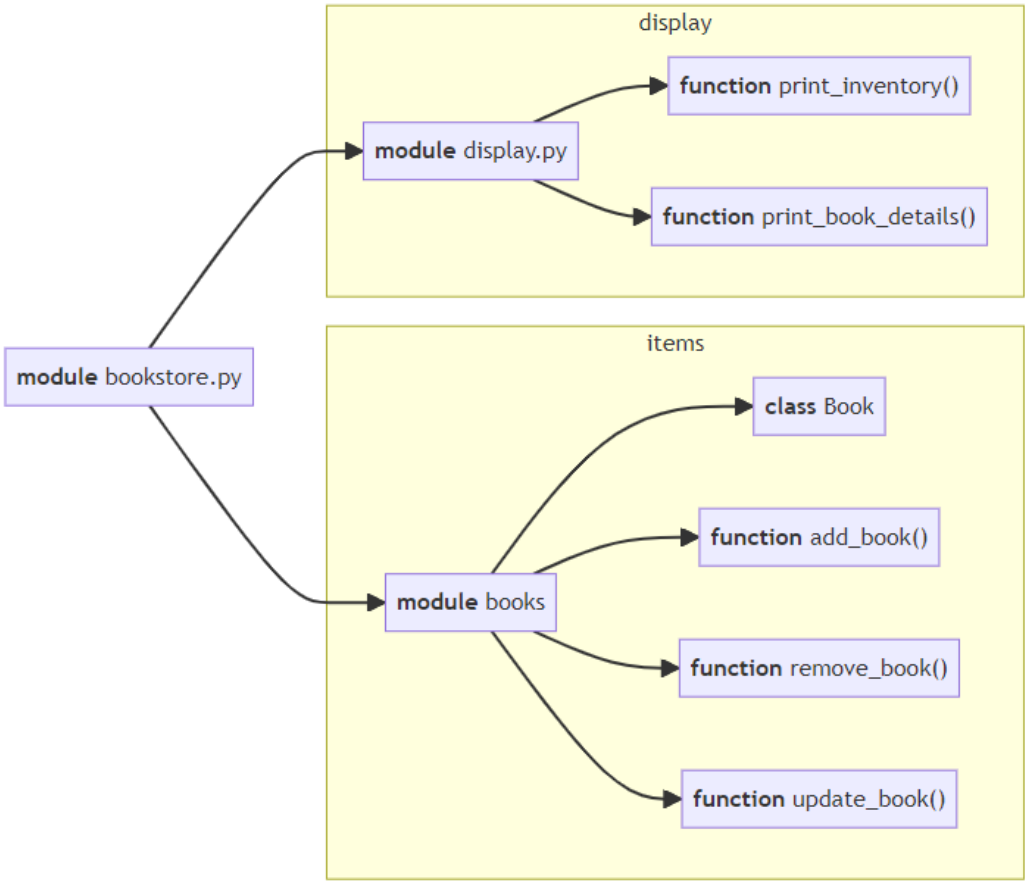


Figure 2.2: Python: Module structure

2.1 | Python project with multiple files

Python code structure is **module-based**, that is, each project contains many `*.py` files, called **modules**, with one module **imports** another.

The project is run from a main script. For example, a user might run the project by entering into the command line:

```
$ python main.py <arguments>
```

and `main.py` would import other modules if needed without any further action. Note that the executed file need not be `main.py`, it can be named arbitrarily.

2.2 | Importing external files

ⓐ | The `import` keyword

To import a module, simply include the line

```
import <module-name>
```

at the beginning of the file. The `<module-name>` is simply the filename, without the `.py` extension.

To use a function in a module, access via the syntax

`<module-name>.<function-name>(<arguments>)`

```
import math                 # math.py imported. This module is located
                            # at the Python/Lib folder that is
                            # accessible by default
g = math.gcd(35, 14)        # function 'gcd' of module 'math' is called
print(g)          # prints out 7
```

Listing 2.1: Python: Using the `math` module

ⓑ | Other ways to import files

`from` keyword

The `from` keyword allows importing a selection (or all) of the `classes` and `functions` from a module.

To include specific `classes` and `functions` from a module, include

```
from <module-name> import <class-names>, <fn-names>
```

at the beginning of the script.

To access all, replace the `classes` and `functions` by the asterisk `(*)`.

```
from <module-name> import *
```

`as` keyword

The `as` keywords allows specifying aliases for `functions` when imported from another module.

```
import math as maths
g = maths.gcd(35, 14)
print(g)          # prints out 7
```

Listing 2.2: Python: Using the `math` module, with module alias

It is common practice to use frequently used modules by their short-hand aliases for better readability, for example, `numpy as np`, `pandas as pd`, etc.

```python
from math import gcd as greatest_common_divisor
g = greatest_common_divisor(35, 14)
print(g)        # prints out 7
```

Listing 2.3: Python: Using the `math` module, with function alias

© | Script mode vs. module mode

Due to the flexible role of a `.py` file, it is important to distinguish if a file is run directly from the `python` command **(script mode)** or via calls from other files **(module mode)**.

Usually the code runs when launched from command, but are not meant to be executed when imported by other files. To deal with the issue, every module has a variable `__name__` that stores either

- `__main__`, if the module is run from the command line, or
- **the name of the importer**, if the module is called from another module.

Therefore, it is common practice to check `if __name__ == __main__` at the beginning of a script, to avoid code run when imported by other modules.

```python
# -- sell.py --
def sell_to(customer):
    print('This book is sold to', customer, '!')

if __name__ == '__main__':
    sell_to('The Owner')
```

```python
# -- main.py --
from sell import sell_to

if __name__ == '__main__':
    sell_to('Everyone')
```

Listing 2.4: Python: Script mode (`main.py`) vs. Module mode (`sell.py`)

```
$ python sell.py
This book is sold to The Owner!

$ python main.py
This book is sold to Everyone!
```

Listing 2.5: Python: Different output at different direct call points

2.3 | Python classes, properties and methods

ⓐ | Declaration

```
class ClassName:
    # class description
    class_var_1 = class_value_1
    class_var_2 = class_value_2

    def __init__(self, value_1, value_2):
        self.var_1 = value_1
        self.var_2 = value_2

obj = ClassName(args...)            # create an instance
```

Listing 2.1: Python: `class` template

```
class Book:
    count = 0

    def __init__(self, name, price):
        self.name = name
        self.price = price

    def print_book_details(self):
        print('Book Title:', self.name, ', Price: $', self.price)
```

Listing 2.2: Python: A simple Python `class` example

ⓑ | Propeties

Propreties (or attributes) are variables associated with an object of the class. They are initialised in the `__init__()` method.

Class attributes are **public** by default, unlike most programming languages with strict OOP paradigm like C++/Java, where variable visibility must be explicitly declared. Instead, by convention:

- variables with a single underscore (`_`) prefix are considered **`protected`**;
- variables with a double underscore (`__`) prefix are considered **`private`**.

Note that these are conventions only, and is not strictly managed by the intepreter.

© | Methods

Methods are functions defined within the class that operate on objects of the class.

- **Instance method**. declared inside a class, must take `self` as an argument. This method acts on an **instance**.
- **Class method**. declared inside a class with the `@classmethod` decorator, must take `cls` as an argument. This method returns a new instance of the class, without having to call the default constructor.
- **Static method**. declared inside a class with the `@staticmethod` decorator, take neither `self` or `cls` argument. This method is independent of the class, the class name sometimes acts more like a method namespace.

```python
class Book:
    count = 0

    def __init__(self, name, price):
        self.name = name
        self.price = price

    def print_book_details(self):
        print('Book Title:', self.name, ', Price: $', self.price)

    @classmethod
    def priceless_book(cls, name):
        return cls(name, 0)

    @staticmethod
    def minimum_price():
        return 2000

    @staticmethod
    def minimum_price_dollars():
        return '$2000'

a_priceless_book = Book.priceless_book("Priceless")
a_priceless_book.print()
print('Minimum price of a valuable book: ', Book.minimum_price_dollars())

# Output:
# Book Title: Priceless, Price: $0
# Minimum price of a valuable book: $2000
```

Listing 2.3: Python: An example of `@classmethod` and `@staticmethod`.

ⓓ | @dataclass

Introduced in Python 3.9, `@dataclass` is a special `class` that automatically implements the `__init__()` and `__repr__()` methods. These are the most common methods for classes that focuses on storing data of a larger object, therefore the `@dataclass` syntax offers a syntactic sugar to the developer, removing the excess boiler-plate code.

```
class Book:
    def __init__(self, name: str, price: float, count: int):
        self.name = name
        self.price = price
        self.count = count

    def __repr__(self):
        print('Book Title:', self.name, ', Price: $', self.price,
              ', Quantity: ', self.count)

book = Book('No Title', 19.99, 4)
print(book)
# Sample Output:
# Book Title: No Title, Price: $19.99, Quantity: 4
```

Listing 2.4: Python: A Python regular `class`

```
from dataclasses import dataclass

@dataclass
class Book:
    name: str
    price: float
    count: int

book = Book('No Title', 19.99, 4)
print(book)
# Sample Output:
# Book(name='No Title', price=19.99, count=4)
```

Listing 2.5: Python: A Python `@dataclass`

Bibliography

[1] Github, “Repositories documentation.”