CEGE0096: Geospatial Programming

Lecture 2: Python and Version Control

October 7, 2019

Aldo Lipani

PYTHON II

Our Python (Imperative) Journey

- Types
- Operators
- Variables
- Simple I/O
- Branching
- Loops
- Functions
- Program Planning
- Strings
- Tuples
- Lists
- Dictionaries
- Files

} Covered in the 1st Lecture

Review Exercise 1

Write a program that requests the user to `input` a temperature in degree Celsius and `print`s out a suitable message according to the temperature state below:

- “Freezing” when the temperature is below zero;
- “Very cold” when it’s between 0 and 10 (not included);
- “Cold” when it’s between 10 and 20 (not included);
- “Normal” when it’s between 20 and 30 (not included);
- “Hot” when it’s between 30 and 40 (not included);
- “Very hot” when it’s above 40.

You are allowed to use variables, operators and if statements.

Function reminder:

`input("message")` prints the message and waits the user for an input

`float` to convert strings to floating-point numbers

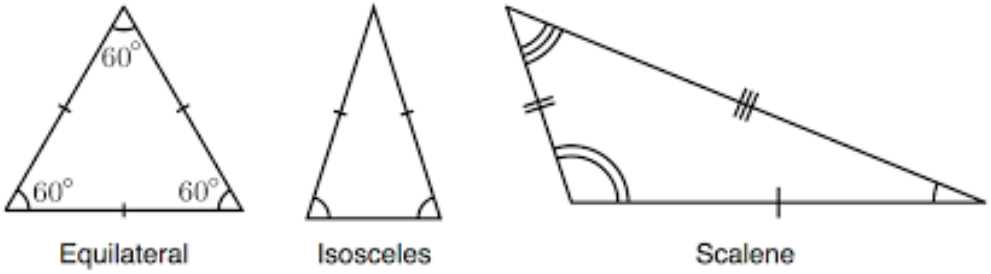
Solution

```
print("Temperature to Perceived Temperature Converter (0.1)")

temp = input("What temperature is outside? ") # This returns a string
temp = float(temp)                            # This converts it to a float

perceived_temp = ""
if temp < 0:
    perceived_temp = "freezing"
elif temp < 10:
    perceived_temp = "very cold"
elif temp < 20:
    perceived_temp = "cold"
elif temp < 30:
    perceived_temp = "normal"
elif temp < 40:
    perceived_temp = "hot"
else:
    perceived_temp = "very hot"

print("Your temperature today is", perceived_temp)
```

Review Exercise 2

Write a program that ask the user for 3 angles in degrees and checks, if possible, which triangle you can build with them:

- Equilateral triangles have all angles equal;
- Isosceles triangles have 2 angles equal;
- Scalene triangles have all of them are different.



Rules are the same of the previous exercise.

Solution

```
print("Angles Checkers for Triangles (0.1)")

angle1 = float(input("1st angle: "))
angle2 = float(input("2nd angle: "))
angle3 = float(input("3rd angle: "))

if angle1 + angle2 + angle3 != 180:
    print("You can't build any triangle out of these angles!!!")
elif angle1 == angle2 and angle2 == angle3:
    print("You can build an equilateral triangle")
elif angle1 == angle2 or angle2 == angle3 or angle3 == angle1:
    print("You can build an isosceles triangle")
else:
    print("You can build a scalene triangle")
```

So far

Can you write a program that prints a number from 1 to 5?

Yes!

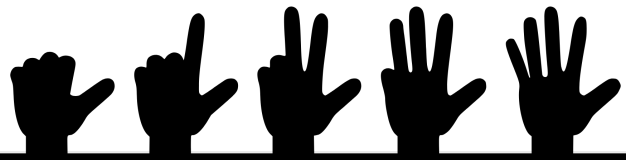
```
print(1)
print(2)
print(3)
print(4)
print(5)
```

Or

```
print(1, 2, 3, 4, 5)
```

Can you write a program that prints a number from 1 to n?

Umh!



While Loop

A `while` loop repeats the code in its block while a condition is verified.

It behaves like an `if` because the block is executed only if the condition is verified. However, unlike the `if` the `while` loop repeats the block.

```
while condition:
    print("hello")
```

```
if condition:
    print("hello")
```

Will this `while` loop terminate?

If the condition is not invalidated by the code in the block of the loop, this program will never terminate.

For Loop

A `for` loop repeats the code in its block for a prefixed number of times.

It is usually used together with a `range` function. This function returns a sequence of numbers starting from 0 that is as long as the integer passed as a parameter.

```
for n in range(5):
    print("hello")
```

Will this `for` loop terminate?

Yes, `for` loops (almost) always terminate!

While vs. For Loops

The `while` loop is more flexible than the `for` loop.

When you know, before the loop starts, how many times the loop needs to repeat a block, then you should use a `for` loop. Use a `while` loop otherwise.

For example, you can write a `while` loop that behaves like a `for`:

```
n = 0
while n < 5:
    print(n)
    n = n + 1
```

```
for n in range(n):
    print(n)
```

Which one would you prefer in this case?

While vs. For Loop (II)

Which one would you prefer when you need…

... to print the letters of a string? `for`

… to compute the average of 5 numbers? `for`

… to compute the average of a user generate sequence of numbers until a “stop” is inputted? `while`

… to iterate over the days of the week? `for`

… to keep asking user generate numbers as long as their sum is equal to 100? `while`

Breaking out of a Loop

Sometimes (**better never**) you may need to break out of a `for` (or `while`) loop even if the prefixed number of iterations has not completed (or if the condition is still true).

To do this you could use the `break` statement

```
for n in range(5):
    if n == 3:
        break
    print(n)
```

```
for n in range(3):
    print(n)
```

The use of a `break` is discouraged because it can cause great confusion when reading the code. If you can avoid it, avoid it!

Continuing a Loop

A `break` statement "jumps out" of the loop.

A complementary statement is `continue`.

This statement allows you to jump to the end of the loop skipping the instructions after the `continue` statement.

```
for n in range(5):
    if n == 3:
        continue
    print(n)
```

```
for n in range(5):
    if n != 3:
        print(n)
```

To be used as often as the `break` statement: almost never!

So far

You now know the basics of Python to solve any problem you may be faced with.

For example, can you count the number of ‘h’s in a string? Sure!

```
string = "Hello, World!"
count = 0
n = len(string)
for i in range(n):
    if string[i] == 'h' or string[i] == 'H':
        count = count + 1
print(count)
```

What if you need to perform this same counting multiple times in your program?

BAD Solution: Copy and paste the same code wherever you need it!

GOOD Solution: Make a function!

Functions

You can define functions when you need code that needs to be used multiple times in your program.

Functions avoid **repetitions** and make the code **reusable**.

You have already seen and used many of them like: `str`, `int`, `float`, `range`, `print` and `input`.

You can `def`ine a new function in any block with an arbitrary number of parameters.

```
def sum(a, b, c):
    return a + b + c
```

The `return` statement is used to indicate what the function should return as an output.

`return` is optional -- if you don't specify `return` the function won't return anything. This does not mean that such a function is useless, e.g. `print` does not return anything. But if specified, **use only one** at the end.

You can name a function as you wish, following the same conventions and restrictions you have for when you name variables.

Functions (II)

You can make optional the specification of some parameters by using default values:

```
def sum(a, b, c = 0):
    return a + b + c
```

In this way, the following two function calls are allowed:

```
sum(1, 2)
sum(1, 2, 3)
```

You can only specify default values starting from the right-most parameter, e.g. this is **not** allowed: `def sum(a, b = 0, c)`

Variables (II) - Scope

The scope of a variable refers to its visibility. A variable is visible when you can access it.

Variables defined within functions are private to the functions. While, variables defined outside functions may be visible within these functions.

```
a = 10
def foo():
    a = 5
    print(a)

foo()
print(a)
```

What does this code print?

```
a = 10
def foo():
    def bar():
        print(a)
    a = 5
    bar()
    print(a)

foo()
print(a)
```

Where is this function visible? (Same rules apply to functions.)

And, this one?

Example: A Function for Counting Hs

Original version:

```
string = "Hello, World!”
count = 0
n = len(string)
for i in range(n):
    if string[i] == 'h' or string[i] == 'H':
        count = count + 1
print(count)
```

Function version:

```
def count_hs(string):
    count = 0
    n = len(string)
    for i in range(n):
        if string[i] == 'h' or string[i] == 'H':
            count = count + 1
    return count
```

You can use this function like this:

```
count_hs("Hello, World!")
count_hs("I'm another string with two more hs!")
```

Program Planning

Python programs are usually wrapped into a main function:

```
def main():
    print("Hello, World!")


if __name__ == "__main__":
    main()
```

The `if` statement is true only if the script is called using a python interpreter, e.g. like this:

```
$ python helloworld.py
```

where the file `helloworld.py` contains the program above.

PEP8: Style Guide for Python Code

PEP stands for Python Enhancement Proposals.

The PEP8 document gives coding conventions for the Python code comprising the standard library in the main Python distribution.

Use **4 spaces** per indentation level.

Surround **top-level function** definitions with **two blank lines**.

' strings and " strings are the same. This PEP does not make a recommendation for this. Pick a rule and stick to it.

About naming variables and functions:

`CamelCase` is discouraged in favour of the **snake_case**.

More details here: https://www.python.org/dev/peps/pep-0008/

Exercise 3 (10 Minutes)

Write a program that **keeps requesting** the user to `input` a temperature in degree Celsius and `print`s out a suitable message according to the temperature state below:

- “Freezing” when the temperature is below zero;
- “Very cold” when it’s between 0 and 10 (not included);
- “Cold” when it’s between 10 and 20 (not included);
- “Normal” when it’s between 20 and 30 (not included);
- “Hot” when it’s between 30 and 40 (not included);
- “Very hot” when it’s above 40.

The program should stop when the user enters a number greater or equal than 100.

You are allowed to use variables, operators, if statements and loops.

Function reminder:

`input("message")` prints the message and waits the user for an input

`float` to convert strings to floating-point numbers

Solution

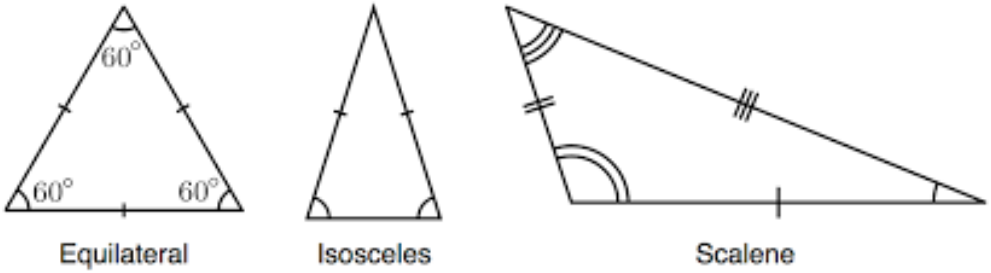
```
print("Temperature to Perceived Temperature Converter (0.2)")

temp = input("What temperature is outside?")
temp = float(temp)

while temp < 100:
    perceived_temp = ""
    if temp < 0:
        perceived_temp = "freezing"
    elif temp < 10:
        perceived_temp = "very cold"
    elif temp < 20:
        perceived_temp = "cold"
    elif temp < 30:
        perceived_temp = "normal"
    elif temp < 40:
        perceived_temp = "hot"
    else:
        perceived_temp = "very hot"

    print("Your temperature today is", perceived_temp)

    temp = input("What temperature is outside?")
    temp = float(temp)

print("Huuu, with such heat I better shut myself down!")
```

Changes with respect to the solution of Exercise 1 are in **bold**!

Exercise 4 (10 Minutes)

Write a program that ask the user for 3 angles in degrees and checks, if possible, which triangle you can build with them:

- Equilateral triangles have all angles equal;
- Isosceles triangles have 2 angles equal;
- Scalene triangles have all of them are different.



This check should be done in a function. The function should return the type of triangle.

Rules are the same of the previous exercise.

Solution

```
def get_triangle_type(angle1, angle2, angle3):
    if angle1 + angle2 + angle3 != 180:
        triangle_type = "none"
    elif angle1 == angle2 and angle2 == angle3:
        triangle_type = "equilateral"
    elif angle1 == angle2 or angle2 == angle3 or angle3 == angle1:
        triangle_type = "isosceles"
    else:
        triangle_type = "scalene"
    return triangle_type

print("Angles Checkers for Triangles (1.1)")

angle1 = float(input("1st angle: "))
angle2 = float(input("2nd angle: "))
angle3 = float(input("3rd angle: "))

triangle_type = get_triangle_type(angle1, angle2, angle3)

if triangle_type == "none":
    print("You can't build a triangle out of these angles!!!")
else:
    print("You can build an", triangle_type, "triangle")
```

Summary

```
def foo(a):
    inp = input("Simon Says type " + str(a))
    while a != int(v):
        print("Wrong answer!")
        inp = input("Simon Says type " + str(a))
```

```
for n in range(5):
    foo(n)
```

1. A function
2. A while loop
3. A for loop
4. The range function
5. A function call

VERSION CONTROL

Version Control

Making regular backups is important.

We have all made the mistake of spending time and effort writing a document and then losing everything when the computer crashes.

Therefore making at least one back up is important, to recover from sudden death.

How do we avoid this?

Version Control (II)

But, how many times does your application allow you to undo?

What happens if you close the document and reopen it?

How do you undo a mistake that you made 1 hour ago and keep everything else you have worked on?

Version Control Software (VCS) provides a system for managing changes to a document or collection of documents.

Files can be reinstated to an earlier snapshot.

Version Control (III)

VCS allows you to have two or more versions of the same program, e.g.:

- The version being developed
- The version that works (used by the client)

VCS enables two or more people to make changes to the same document. The changes can be later merged together.

Notable VCSs are:

Git

SVN

Which one should we use? It really depends on your needs.

Git

Git is the most widely used VCS

Developed by Linus Torvalds (the creator of Linux)

Git is a **distributed** VCS (the complete history of your project is in every copy of the repository). While, SVN is centralized.

How Does Git Work?

Git can be setup in a directory of your choice. All files (and paths) are managed by Git.

Git creates a hidden sub-directory in which all metadata and snapshots are saved. This is the **repository**.

Git does not need to be online. All commits are saved in the local directory.

The **local** repository can be synchronized with a **remote** repository.

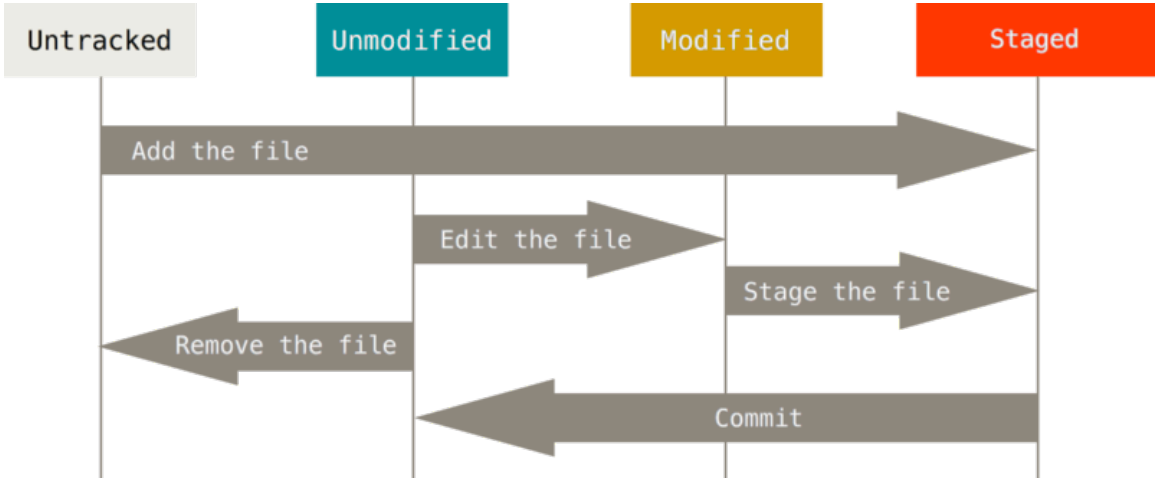
The **working directory** is a checkout of one version of the project.

The **staging area** (or index) is a file that stores information about what will go into your next commit.

The Four States of a File

In Git a file can be in one of the 4 following states:

1. **Untracked:** This file is not under VCS;
2. **Modified**: You have modified the file but have not committed to your database yet;
3. **Staged:** You have marked a modified file in this current version to go into your next commit;
4. **Committed:** the data is safely stored in your local database.

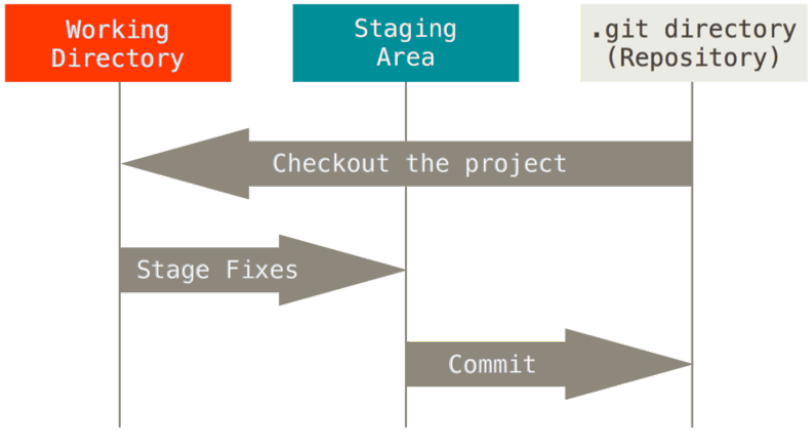


Basic Workflow

1. You modify files in your **working directory**.

2. You selectively **stage** just those changes you want to be part of your next commit, which adds **only** those changes to the **staging area**.

3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.



Installing Git in Anaconda

From your `geospatial` anaconda environment, type:

```
$ conda install git
```

And, follow the instructions.

To check if you have correctly installed Git, type:

```
$ git --version
```

This should output something like this:

```
git version 2.20.1
```

Getting a Git Repository

You can get a git repository in two ways:

- You can take a local directory that is currently not under VCS, and turn it into a Git repository, or;
- You can `clone` an existing Git repository from elsewhere.

To create a git repository in a folder, type:

```
$ git init
```

You can always add later a remote repository typing:

```
$ git remote add origin <url>
```

To `clone` an existing repository (it will create the folder for you), type:

```
$ git clone <url>
```

Checking the Status of Your Files

You can check the status of your files by typing:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

This means you have a clean working directory.

If you add a `NEW_FILE` and type:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
   (use "git add <file>..." to include in what will be committed)

     NEW_FILE

nothing added to commit but untracked files present (use "git add" to track)
```

Tracking New Files

You can add this `NEW_FILE` to the tracked files (staged) by typing:

```
$ git add NEW_FILE
```

Now if you rerun:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed: (use "git reset HEAD <file>..." to unstage)
    new file: NEW_FILE
```

You can tell that the `NEW_FILE` is staged because it's under the "Changes to be committed" heading.

You can remove files by using:

```
$ git rm OLD_FILE
rm 'OLD_FILE'
```

Staging Modified Files

If the file was already tracked and you just modified it. And you run:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master’.
Changes to be committed:
    (use "git reset HEAD <file>..." to unstage)

     new file: NEW_FILE Changes not staged for commit:

    (use "git add <file>..." to update what will be committed)
    (use "git checkout -- <file>..." to discard changes in working directory)

     modified: MODIFIED_FILE
```

The `MODIFIED_FILE` is not yet staged. It means that if you commit now this file won’t be added to the new snapshot.

To add it to the staged ones, type:

```
$ git add MODIFIED_FILE
```

Committing your Changes

After you have added all the files that you want to commit into your stage area, then you can type:

```
$ git commit -m ”fix: fix for method is_intersecting method of the object Polygon"
[master 463dc4f] fix: fix for method is_intersecting method of the object Polygon
2 files changed, 2 insertions(+)
create mode 100644 geometry.py
```

To view the commit history, type:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Aldo Lipani <aldo.lipani@ucl.ac.uk>
Date: Mon Mar 17 21:52:11 2008 -0700

 fix: fix for method is_intersecting method of the object Polygon

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Aldo Lipani <aldo.lipani@ucl.ac.uk>
Date: Sat Mar 15 16:40:33 2008 -0700

  feat: added geometry models
```

Undoing Things

If you commit to early, you can undo the commit and go back to staged area, typing:

```
$ git commit --amend
```

If you want to unstage a file

```
$ git reset HEAD FILE_NAME
```

To revert a file to a previous version type:

```
$ git checkout -- FILE_NAME
```

Working with Remote

To get data from the remote you can run

```
$ git fetch origin
```

This command goes out to the remote project and pulls down all the data from the remote project that you don't have yet.

To also update your working directory use pull:

```
$ git pull origin
```

When you have your project at a point that you want to share, you can push it to the remote:

```
$ git push origin master
```

Note: This command works only if you have writing access to the repository.

Ignoring Files (.gitignore)

Sometimes you want to ignore files that you know you will never need to commit.

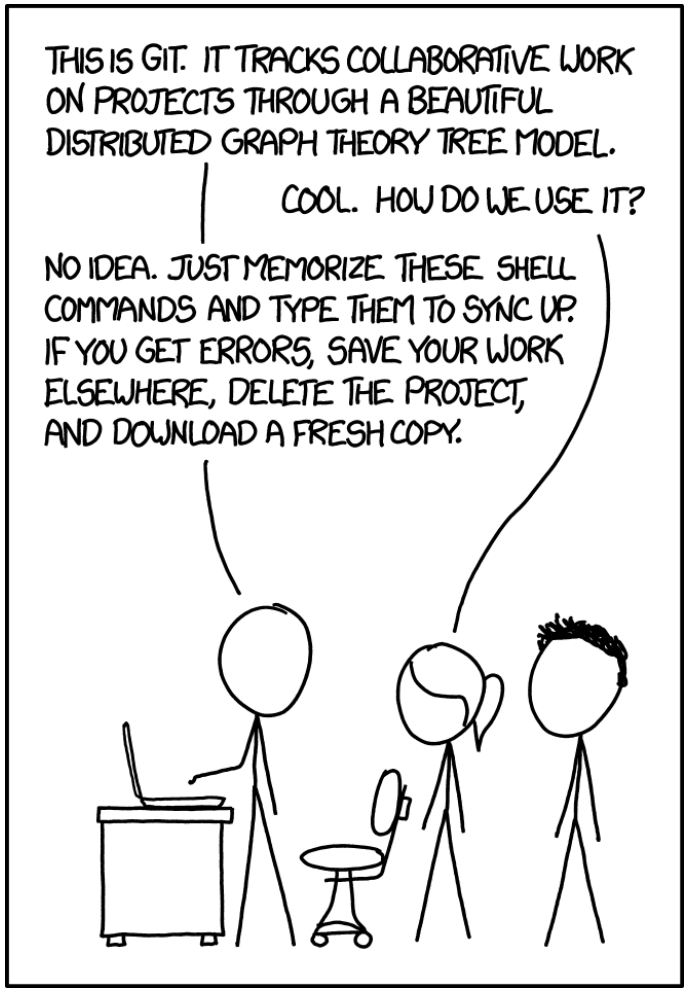
These files could be OS hidden files, IDE folders, compiled files, etc.

You can ignore them by using a special file called `.gitignore` (the file name must contain the dot).

You need to create this file.

This file is simply a list file names.

Git at the End of the Day



VCS Best Practices

Do not push code that crushes! It will annoy your colleagues to find out (usually in the morning) that the code crashes in a point where it was not supposed to crash.

Write meaningful commit messages!

Start your commit with:

- **fix:** when you are fixing a bug;
- **feat:** when you are introducing a new feature;
- **BREAKING CHANGE:** when you are making change (rare);
- Other prefixes are allowed.

Here you find more details: https://www.conventionalcommits.org/en/v1.0.0/

GitHub

GitHub is provides hosting for software development version control using Git.

Do not confuse:

- Git -- The VCS;
- GitHub -- A repository hosting service.

Git users can use GitHub to host their remote repositories.

Reference about Git

The entire Pro Git book, written by Scott Chacon and Ben Straub and published by Apress, is available here. All content is licensed under the Creative Commons Attribution Non Commercial Share Alike 3.0 license. Print versions of the book are available on Amazon.com.

https://git-scm.com/book/en/v2