Introduction to Python

INFX 598: Intro to Programming

Joel Ross
Spring 2017

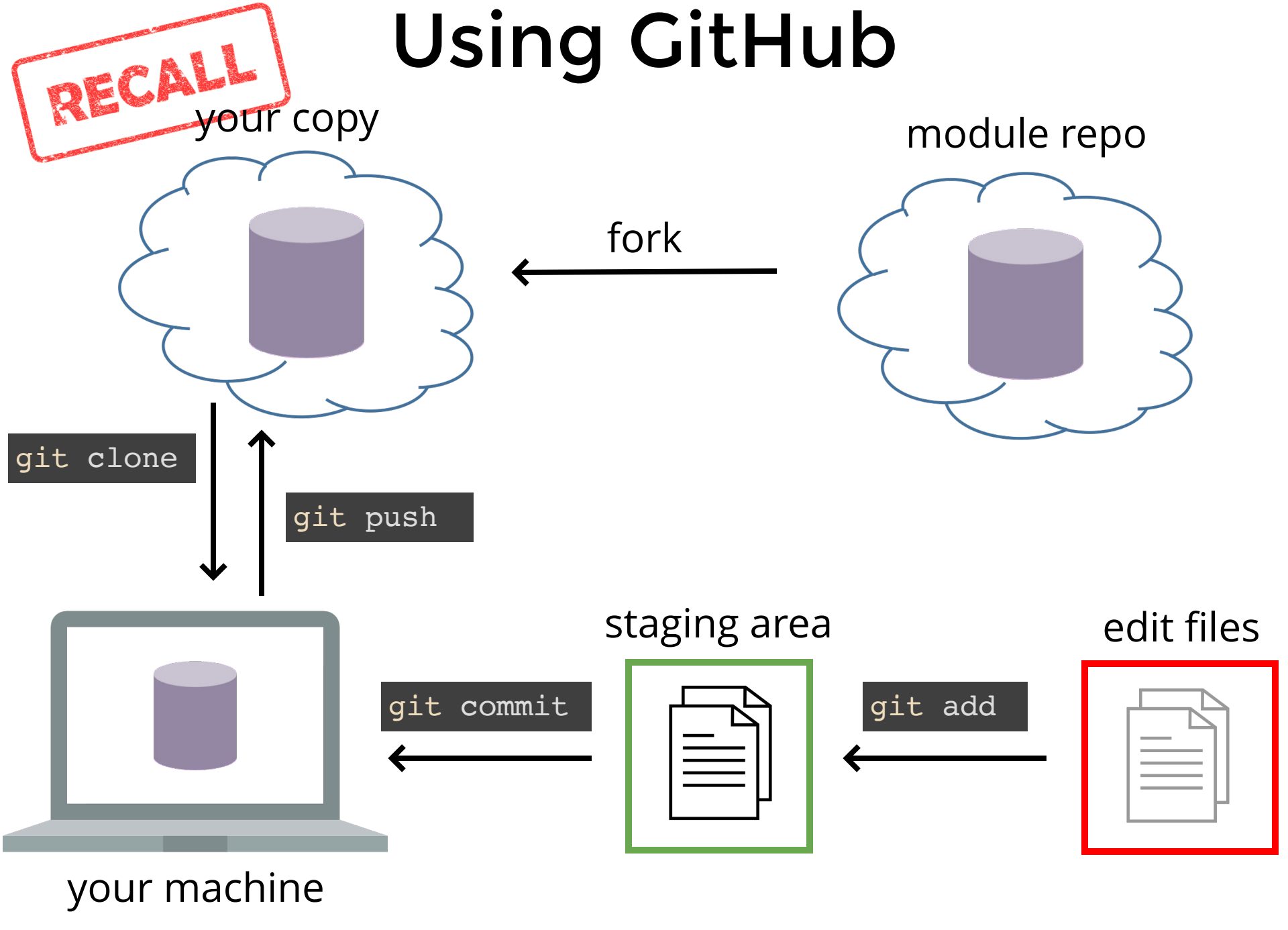
Today's Objectives

By the end of class, you should be able to

- Feel confident with version control basics.
- Understand how to to **write**, **execute**, and **debug** computer programs in the Python language
  - launch and interact with **Jupyter Notebooks**
- Store data in **variables**
- Utilize **functions** to manipulate data

Any questions on the homework?

Using GitHub



Warmup: Git and CLI

1. **Fork and clone** module5 to your computer (pick a general directory where you'll keep all your class work: `Documents`, `Desktop`, `~` (Home), etc).

2. Change directory into the module repo

3. Use the command-line to create a file `STUDENT.md` that contains your name:

   ```
   # Student
   These exercises were completed by YOUR NAME.
   ```

   Hint: use `echo` and redirects (`>>`) to write the content!

4. `add` and `commit` your changes

5. `push` your changes back to GitHub, and view the file on the website to confirm that everything worked!

   ```
   # change remote bookmark
   git remote set-url origin https://github.com/USER_NAME/module5-python-intro
   ```

python™

A high-level, general purpose, interpreted programming language

- ***High Level***: closer to human language than machine language

- ***General Purpose***: can be used for multiple domains

- ***Interpreted***: Python language is translated into machine language as it is being run (vs. *compiled language*)

Python Versions

Python originally released in **1991**

Python 2 released in **2000**

Python 3 released in **2008**

Python 2 and Python 3 are not compatible!

We'll use Python 3

- Easier to learn: fewer "gotchyas"
- Easier to convert from 3 to 2 than from 2 to 3
- Python 3 has additional features
- Python 3 is the future!

```
# Check Python version on the command-line (Mac)
python --version

# Windows (using Git Bash default config)
winpty python --version

# on lab machines (Mac), need to switch to Python 3
# do this once (per terminal)
source activate python3
```

Command-Line Python

It is possible to run the **Python interpreter** from the command-line. This will let you specify commands in the **Python** language for the computer to *interpret* and *execute*.

python shell is easy to test some codes

joelross — python — 80×15

~ — python

```
Last login: Wed Mar 29 15:06:09 on ttys007
is-joelrossm13x:~ joelross$ python
Python 3.6.0 |Anaconda 4.3.1 (x86_64)| (default, Dec 23 2016, 13:19:00)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 5+9+8
22
>>>
```

We can also run **Python Scripts** (`.py` files containing Python code) from the command-line.

Jupyter Notebooks

Jupyter Notebooks are *interactive web applications* that can be used to write and execute Python programs (as well as textual content). Notebooks can also be shared with others!

Using Jupyter

Start the Jupyter Notebook server (command-line)

```
jupyter notebook
```

Create a new notebook or open an existing one (`.ipynb`)

Type your code into a *code cell*.

won't run previous code

- `shift + enter` to execute a cell (or use the **Cell** menu)

You can add additional cells, including *Markdown cells*.

- See the **command palette** for more options 

You can "restart" the interpreter through the **Kernel** menu.

Python Syntax

Comments

Text that is **not** read by the interpreter.

Anything after a `#` is skipped (until the end of the line).

Used to give more information to the **human**.

```
# Add some numbers
5+9+8  # 22
```

Comments should include information that is **not** otherwise in the program.

Include lots of comments!

Printing

Use `print()` to print whatever is in the parentheses to the console. This is an example of a **function.**

```
# My first program
print("Hello world!")
```

What could go wrong?

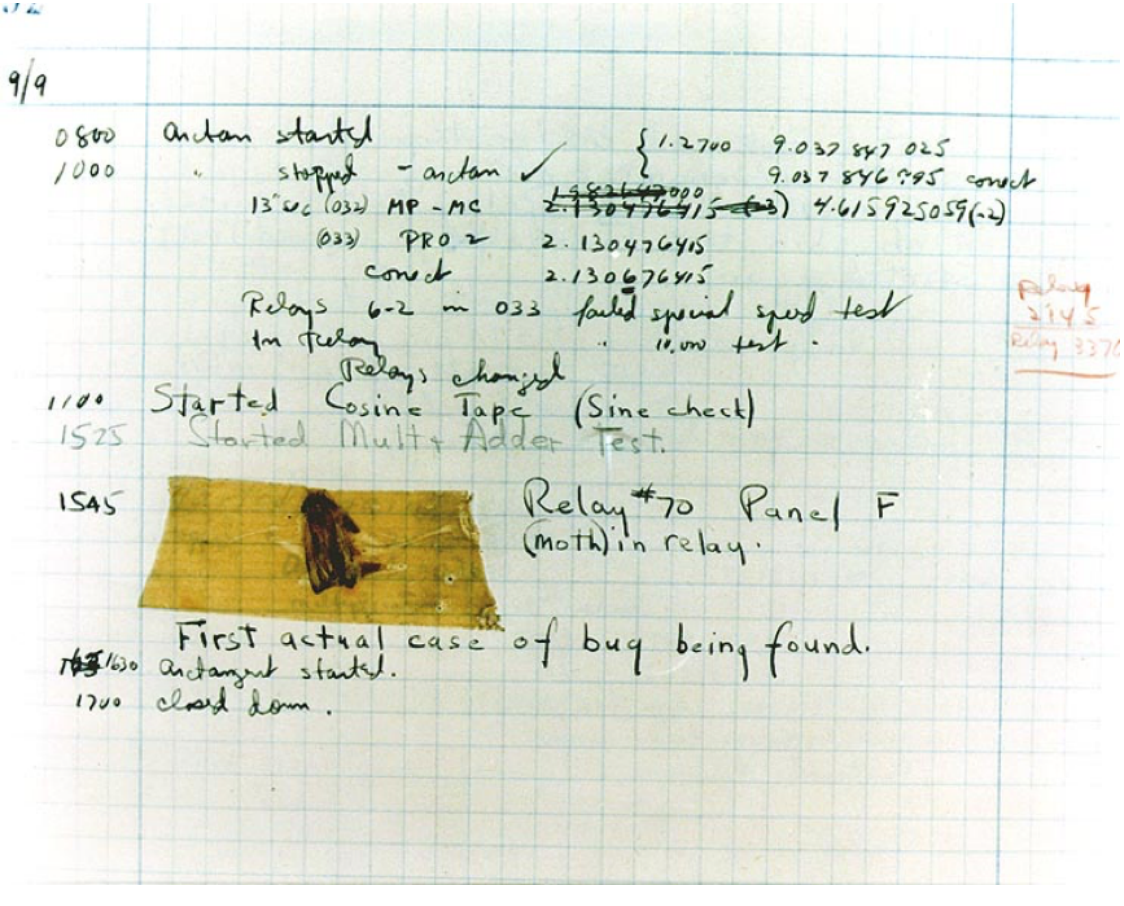
```
# My first program
print "Hello world!"
```

Python 2 syntax!

```
# My first program
print("Hello world!)
```

```
# My first program
print("Hello wold!")
```

Computer Bugs



First computer bug (1946)

Admiral Grace Hopper

Errors

Syntax Error

- An error in the use of the Python language. A problem with how you said something.
- Interpreter will error at the site of the problem.

Logical Error

- An error in the algorithm you used. A problem with what you said to do.
- Interpreter will error, but possibly <u>after</u> the problem.

Semantic Error

- An error in your approach to solve a problem.
- Interpreter will not error, but will not do what you want.

Variables

RECALL

$$y = x^2 + 3x + 7$$

Variables

A label that refers to a **value** (data)

assignment

```
my_num = 598
```

variable
(label)

value
(data)

Using Variables

Once we have a variable, we can use that label to **refer** to a value (in place of a value)

```
x = 4  # x refers to 4
y = x  # y refers to the value of x (4)
z = y + 7  # y refers to sum of y and 4 (11)

z = z + 1  # take z, add 1, and store result
           # back in z
```

*When a label is on the **left**, it means the **variable**.*

*When a label is on the **right**, it means the **value**.*

Assignment is not Equality!

Assignment ***gives a label*** to a value. Changing which value a variable refers to doesn't change other variables.

```
x = 3
y = 4
x = y  # assign the value of y (4) to x
print(x)  # 4
y = 5
print(x)  # 4;
```

Good variable names describe the data

```
# valid program, but what does it mean?
a <- 35.0
b <- 12.50
c <- a*b

# much better!
hours <- 35.0
pay.rate <- 12.50
earnings <- hours * rate

# Get out.
x1q3z9ahd <- 35.0
x1q3z9afd <- 12.50
x1q3p9afd <- x1q3z9ahd * x1q3z9afd
```

Data Types

Variable Types

All *values* in Python have a particular data type or **class** ("classification"). Class determines how that data can be used.

```
type(7)  # <class 'int'> (integer)
type(3.14)  # <class 'float'> (decimal
type("Hello")  # <class 'str'> (string, text)
```

Python is **dynamically typed**, so v*ariables* (labels) can refer to any type of value.

```
x = "Hello"  # value is a string
x = 42  # value is now an integer
```

Numbers

Integers (whole numbers) and floats (decimals) are used to store numbers.

```
x = 2  # whole number
y = 3.5  # decimal (floating point) number

# can perform mathematical operations
z = x + y
```

operator operands

```
# Use parentheses to enforce order of operations
z = 3*(x-y)**2
```

If an operand is a `float` *or* you use the division (`/`) operator, the result will be a float. // return intr

Strings

Strings of characters (letters, punctuation, symbols, etc).
Written in single or double quotes.

```
my_name = "Joel"  # 'Joel' would be equivalent

# Can include any keyboard symbol (and more!)
course = "Infx 598: Intro to Programming!"

# can concatenate strings together
greeting = "Hello" + " " + "World"
```

string can be muliplied by number,
string cant be multiplied by string

\n go the next line
\t tab

Module 5 exercise-1

Functions

RECALL

$$f(x) = 4x - 3$$

function name

input (domain)

result (range)

Functions

A named sequence of instructions (lines of code). We **call** a function to do those steps.

```
print("Hello world")
```

function name argument (value)

Functions ***abstract*** *computer programs!*

Functions

Function **arguments** are the "inputs".

```
# prints "Hello+++World"
paste("Hello", "World", sep="+++")
```

multiple arguments are separated by commas

keyword argument

```
# rounds 5/7 to the nearest .01
round(5/7, 2)  # 0.71
```

Argument order (position) usually matters

expressions in args are evaluated **before** the function is executed

Functions

Functions may **return** a value (the "output"). This value must be *stored in a variable* for the machine to use later!

```
# store min value in smallest_number variable
smallest_number = min(1, 6/8, 4/3, 5+9)  # 0.75

# use the variable as normal, such as for math
twice_min = smallest_number * 2  # 1.5

# use functions directly in expressions
# (the returned value is anonymous)
number = .5 * round(9.8)  # 5.0

# pass the result of a function as an arg to another!
# watch out for where the parentheses close!
print(min(2.0, round(1.4)))  # prints 1
```

Object Methods

All data values in Python are **objects**, which include both data (attributes) and behaviors (methods, a.k.a functions)

We can call methods (functions) **<u>on</u>** a particular data value in order to apply that behavior *to that value*. (e.g., tell *a particular* `person` *variable* to `say_name()`).

Dot Notation

We call a method on a data value by using **dot notation**, putting the data value, then a dot (`.`), then the function call.

```
message = "Hello World"

# call the lower() method on the message
# original string does not change
lower_message = message.lower()  # "hello world"

# call the replace() method on the message
western_message = message.replace("Hello", "Howdy")
## "Howdy World"
```

*The dot is like an **'s** in English: "execute `message`'s `lower()`"*

Module 6 exercise-1

Built-in Functions

https://docs.python.org/3/library/functions.html

String Methods

https://docs.python.org/3/library/stdtypes.html#string-methods

Modules

Python functions are organized into **modules**, which need to be individually loaded. This helps reduce memory usage (only for functions you actually need).

```
# load the math module (contains math functions)
import math

# call the math module's sqrt() function
math.sqrt(25)  # 5.0, (square root of 25)

# print out the math module's `pi` variable
print(math.pi)  # 3.141592653589793

# import specific function, available globally
from math import gcd

# call gcd function (greatest common denominator)
gcd(56,42)
```

Module 6 exercise-2

Action Items!

- Be comfortable with **module 5 & 6** by Thu
- Assignment 1 due ***tonight***!
- Assignment 2 due next Tuesday

Thursday: writing functions