MPCS 51042-2: *Python Programming*

Week 1: Types, operations, statements, and syntax

September 25, 2017

Course Logistics

Outline

Week	Date	Topics	Assignments Due
1	Sep. 25	Types, operations, statements, and syntax	–
2	Oct. 2	Functions, generators, and scope	Assignment 1
3	Oct. 9	Classes and object-oriented programming	Assignment 2
4	Oct. 16	The Python data model	Assignment 3
5	Oct. 23	Decorators and dynamic attributes	Assignment 4
6	Oct. 30	Packaging and distribution (Midterm)	Assignment 5
7	Nov. 6	Testing, debugging, logging, documentation	–
8	Nov. 13	Scientific computing stack	Assignment 6
9	Nov. 20	Parallelism and concurrency	Assignment 7
10	Nov. 27	Special topics	Assignment 8
11	Dec. 4	Final Exam	–

Policies

- **Late policy**: You have one week to complete assignments. No late assignments will be accepted.
- **Academic honesty**
  - Do not ask another student for their code
  - Do not show/share your code with another student
  - Do not post your code in a publicly-accessible manner
  - Do not use code you find on the internet, in a book, etc.
- **Course grade**
  - Homeworks: 60%
  - Midterm: 15%
  - Final: 25%

Grade boundaries

- 95–100: A
- 90–95: A-
- 85–90: B+
- 80–85: B
- 75–80: B-
- 70–75: C+
- < 70: Dealt with on a case-by-case basis

Discussion

Two options for questions/discussions:

- Canvas – main LMS for UChicago
- Slack – `https://join.slack.com/t/mpcs51042/signup` (make sure you use @uchicago.edu email)
- Rather than emailing questions to the teaching staff, please post your questions on Canvas/Slack.

Submitting Assignments

- GitHub will be used for submitting assignments
- Sign up for an account on GitHub and let us know your username
- For each assignment, you'll receive a link that allows you to setup a private repository on the uchicago-python organization
- Commit your solution to the repository by the due date

Staff

Role	Person	Office Hours	Email
Instructor	Paul Romano	After class	romano@uchicago.edu
TA	Ron Rahaman	TBD	rahaman@uchicago.edu
Grader	TBD	—	

Suggested References

O'Reilly books (available electronically via Safari):

- Learning Python, by Mark Lutz
- Fluent Python, by Luciano Ramalho
- Python in a Nutshell, by Martelli, Ravenscroft, and Holden
- Python Cookbook, by Beazley and Jones (free to all)

Other:

- Python Essential Reference, by David Beazley

Introduction

Goals

1. Become comfortable and confident in writing Python code
2. Be equipped with enough knowledge to understand code written in Python
3. Learn the "idioms" of the language
4. Understand the Python ecosystem: language, standard library, third-party packages
5. Develop software development skills: version control, testing, documentation

Python

High-level, general-purpose programming language. Often described as:

- A scripting language
- An interpreted language
- Procedural, object-oriented, and functional

Tradeoffs

Pros

- Ease of programming
- Readability
- Expressiveness
- Rich standard-library
- Third-party packages
- Cross-platform
- Time to useful development

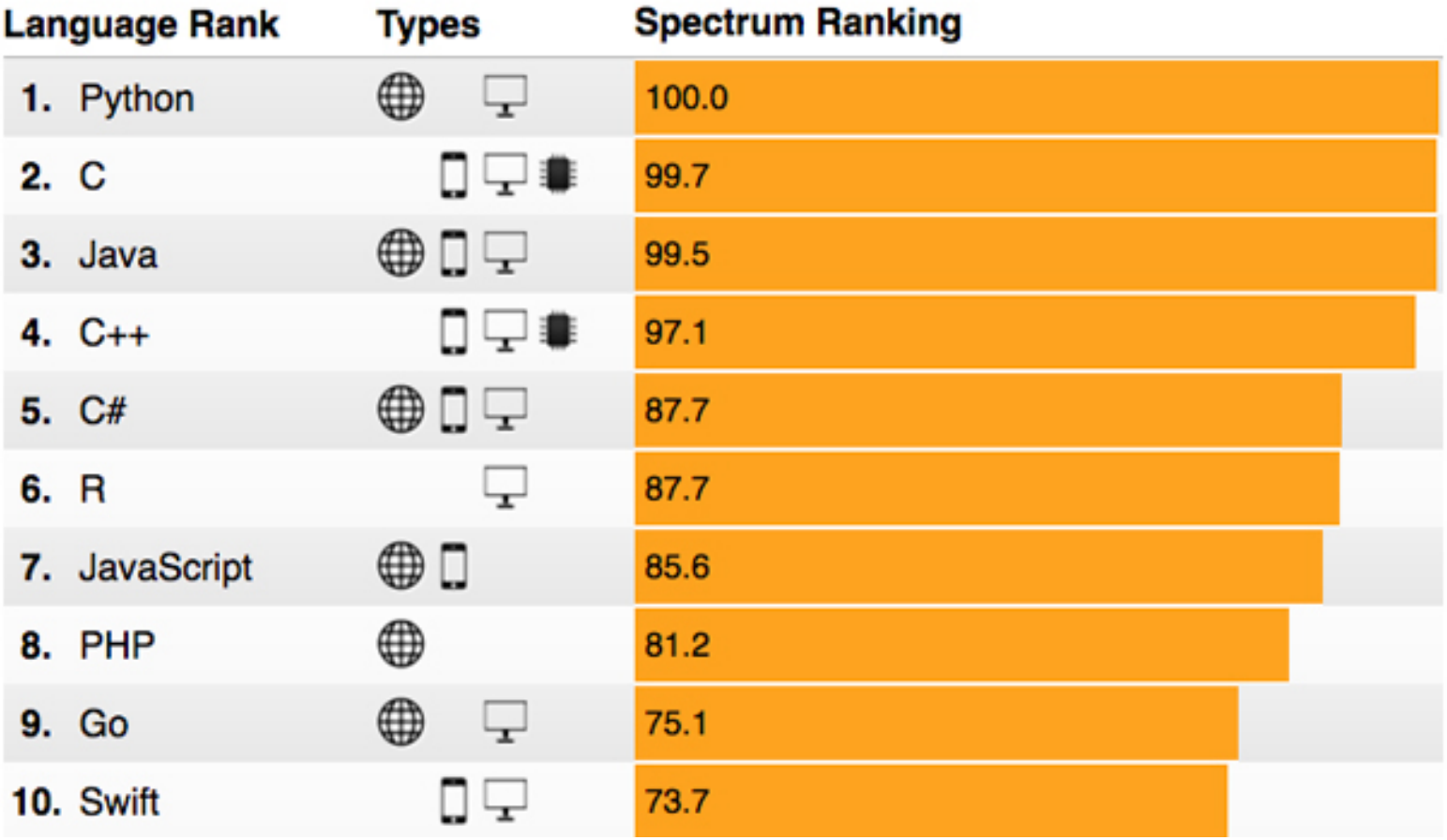
Cons

- Performance
- Parallelism can be tricky
- Reliance on interpreter
- Changing language

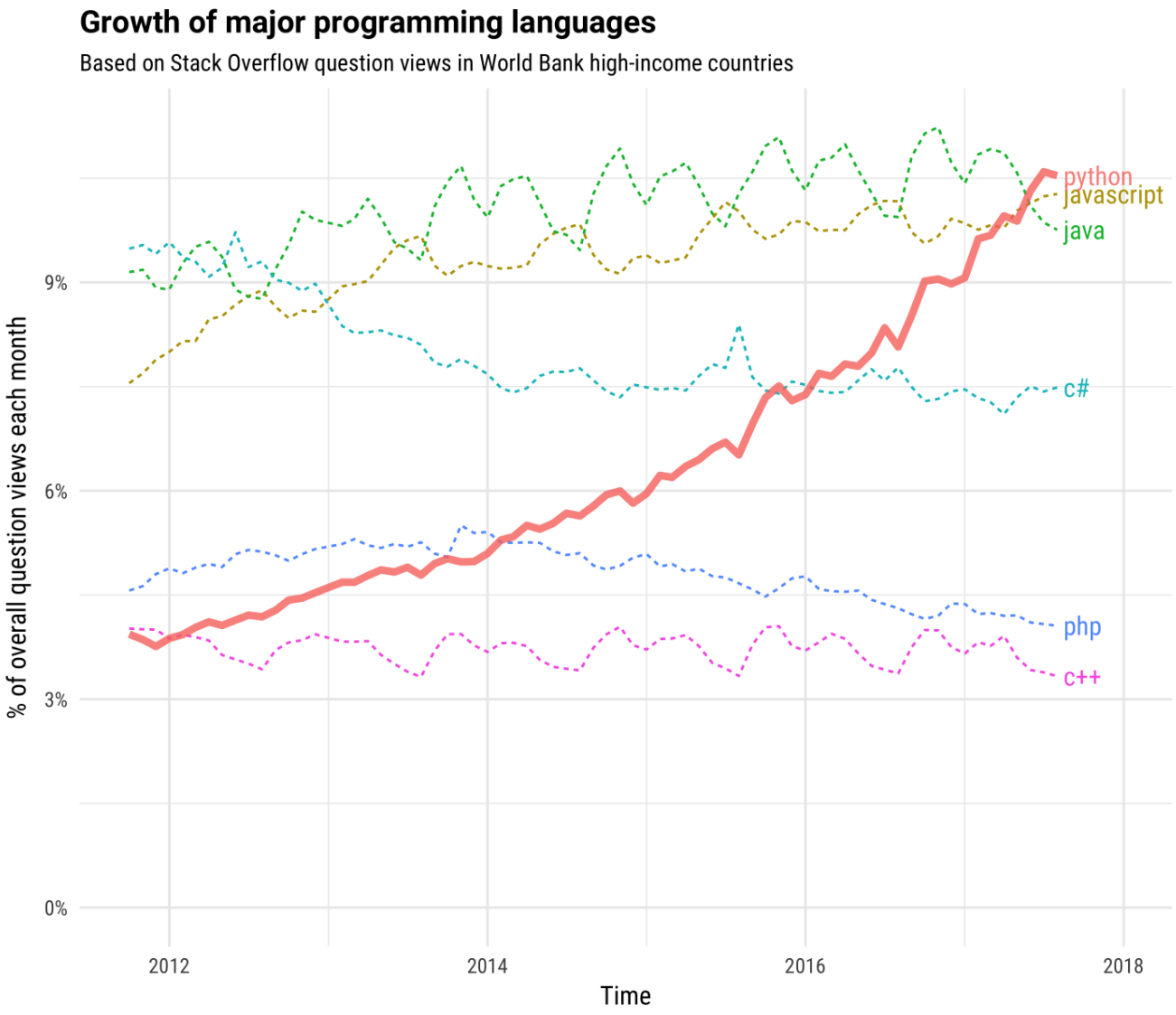
What is Python used for?

- Web development
- Rapid prototyping
- Databases
- Scientific computing
- Systems programming
- Data mining
- Gaming
- ...

IEEE Spectrum 2017



Python growth



Source: The Incredible Growth of Python

Your first program in Python

```
print("Hello, world!")
```

The Python REPL

REPL = read, eval, print, loop

```
>>> print("Hello, world!")
Hello, world!
```

Alternative Python REPLs

- IPython
- ptpython
- Jupyter notebook

Many online REPLs as well:

- repl.it
- pythonanywhere
- trinket
- CodeSkulptor

Compiled vs. Interpreted

- What does it mean for a language to be compiled?
- What does it mean for a language to be interpreted?

What happens when you run python?

Language vs. Implementation

Is Python compiled?

- Most implementations compile to bytecode/machine instructions

Is Python interpreted?

- Most implementations have a bytecode interpreter, but some perform JIT compilation instead (Jython, PyPy)

Python 2 vs. Python 3

- There are two major versions of Python: 2.7 and 3.x
- Python 3 was originally released in Dec. 2008 but hasn't had serious uptake until now
- Python 2 end-of-life was originally scheduled for 2015, but then was moved back five years to 2020
- As a professional programmer, you are better off knowing Python 3
- We will focus solely on Python 3, with occassional notes about differences between 2 and 3

Python Implementations

- **CPython**
- PyPy (implemented in RPython)
- IronPython (implemented in .NET)
- Jython (implemented in Java)
- Stackless
- Skulpt (implemented in JavaScript)
- Grumpy (Python transcompiled to Go)
- MicroPython (for microcontrollers)

PEPs

- PEP = Python Enhancement Proposal = formalized way for someone to propose change in the Python language
- Follows typical open source model:
  - Discuss idea on python-ideas@python.org
  - Create a fork of the Python PEPs repository
  - PEP editors review proposal for structure, formatting, errors
  - Ultimate approval must come from Python's BDFL (Guido van Rossum)

Basic syntax and types

Syntax of Python

```
import sys    # Load a library module

x = 3
y = 4
print('hello world')   # Say hello
print(x**y)            # Raise x to the y-th power
print(sys.platform)    # Show what platform we're on

if sys.platform == 'win32' or sys.platform == "cygwin":
    print("Let's do something Windows-specific")
```

Notes about syntax

- Program consists of multiple lines of statements
- Statements are separated by newline characters
- `import` binds a name to a library module
- Assignment statements binds a name to the result of an expression
- Expression statements simply evaluate one or more expressions (often calling a function like `print()`)
- Comments are denoted by #
- Case sensitive
- Automatic memory management

Attribute access

- The . notation is used to access attributes of objects

```
# Using . to access module variables
import sys
print(sys.platform)
x = sys.exc_info()

# Using . to access methods of an object
s = 'Hello'
t = s.upper()
```

Where are the {braces}?

Python uses leading whitespace to determine the indentation level, which in turn determines the grouping of statements.

```
def binary_search(seq, t):
    min = 0
    max = len(seq) - 1
    while True:
        if max < min:
            return -1
        m = (min + max) // 2
        if seq[m] < t:
            min = m + 1
        elif seq[m] > t:
            max = m - 1
        else:
            return m
```

You are not allowed to mix tabs and spaces in Python 3!

Language vs. Library

Core language

- Statements
- Built-in types
- Built-in functions
- Exceptions
- User-defined functions/classes

Standard library

- `re` — regular expressions
- `collections` — container datatypes
- `math` — mathematical functions
- `random` — pseudo-random numbers
- ...

Built-in types

- Boolean: `bool`
- Numeric types: `int`, `float`, `complex`
- Text sequence: `str`
- Sequence types: `list`, `tuple`
- Set types: `set`
- Mapping types: `dict`

Numeric types

- Integers have unlimited precision
- Floating point numbers are implemented using `double` in C
- Mixed arithmetic is fully supported (narrower type is "widened")
- Operations: +, -, *, /, //, %, **
- Functions: `abs`, `math.sqrt`, `math.exp`, `math.cos`, ...
- Explicit casts with `int()`, `float()`, `complex()`
- Complex numbers formed with a `j` suffix

Numeric literals

Integer literals:

```
-17
79228162514264337593543950336 # Unlimited precision
100_000_000_000  # Underscores are new in Python 3.6
0o377            # Octal literal
0xdeadbeef       # Hexadecimal literal
0b_1110_0101     # Binary literal
0B101010
```

Floating point literals:

```
3.14
10.
.001
1e100
3.14e-10
0e0
3.14_15_93
```

Booleans

- Case-sensitive constants: `True` and `False`
- Result from comparison operators: <, <=, >, <=, !=, ==, `is`, `is not`,
- Used in `if` and `while` loops
- Logical operators: `and`, `or`, `not`
- Ternary operator: `x if b else y`

Strings

- `'...'`, `"..."`, `"""..."""`
- Methods: `upper()`, `find()`, `strip()`, `split()`, ...
- Get length with `len()`
- Can add, multiply strings
- Use `str()` to convert other types
- Check for substrings with **`in`** operator

Sequence types

- List: an ordered, mutable collection of objects
  - Methods: `append()`, `insert()`, `reverse()`, `pop()`, `sort()`, ...
  - Construct with `list()` or `[...]`
- Tuple: an ordered, immutable collection of immutable objects
  - Construct with `tuple()` or `(...)`
- Can get length with `len()`
- Indexed from 0
- Negative indexing can be used to count from end
- Can refer to "slices"

Sets

- Unordered collection of distinct (hashable) objects
- Construct with `set()` or `{'s', 3, 2.0}`
  - However, `{}` is not a set
- Methods: `add()`, `difference()`, `intersection()`, `union()`
- Can also use operators: `in`, `&`, `|`, `-`, `^`
- Get length with `len()`

Dictionaries

- Mapping or associative array
- Indexed by *key* rather than by a range of numbers
- Construct with `dict()` or `{'a': 1, 'b': 2, ...}`
- Set item:

```
person = {'eyecolor': 'blue', 'age': 15}
person['name'] = 'John'
print(person['age'])
```

- Iterate with `keys()`, `values()`, `items()`
- Other methods: `get()`
- Delete a key with **`del`**
- Get length with `len()`
- **`in`** operator compares against keys, not values

if statement

- Execute block of code if a condition is satisfied
- Basic syntax:

```
if condition:
    statement 1
    statement 2
    ...
elif condition2:
    statement 3
    ...
else:
    ...
```

- Lines below the **if** must be indented the same

Truthiness

- The **`if`** statement expects a Boolean, so anything that is not Boolean is implicitly cast to Boolean with `bool(...)`
- For sequence types, having length zero is considered **`False`**
- Any of the following are considered **`False`**: **`None`**, 0, empty sequence or mapping

while statement

Repeat block of code as long as condition is true:

```
while condition:
    statement 1
    statement 2
    ...
```

for statement

- Iterate over the elements of a sequence:

```
for var in sequence:
    statement 1
    statement 2
    ...
```

- `list`, `set`, and `dict` are all sequences that can be iterated over
- Need to be careful with dictionaries

Integer sequences

- Special `range` type that represents integer sequences
- For example, all integers from 0 to 100 in intervals of 4

```
for x in range(0, 100, 4):
    print(x)
```

- Note that `range` is actually its own type, not just a function that creates a list/iterable

Multiple assignment

- We can actually assign multiple objects in a single statement

```
x, y = (1, 2)

# Nesting is permissible
(x, y), z = [(3, 4), 'a']
```

- RHS side can be any (nested) iterable
- Usually used to "unpack" iterables in a **for** loop

```
>>> L = [(1, 2), (3, 4), (5, 6), (7, 8)]
>>> for a, b in L:
...     print(a, b)
...
1 2
3 4
5 6
7 8
```

Iteration functions

`zip` creates a list of tuples

```
numbers = [3, 2, 5, 2]
words = ['this', 'class', 'is', 'cool']
for num, word in zip(numbers, words):
    print(num, word)
```

`enumerate` gives an index of iteration

```
for i, word in enumerate(words):
    print('Word {} is {}'.format(i, word))
```

Working with files

```
# Read two lines of a file
f = open('somefile', 'r')
x = f.readline()
y = f.readline()
f.close()

# Write a file
g = open('newfile', 'w')
g.write('Python is cool.\n')
g.write('And so is this class.\n')
g.close()
```

Iterating over files

Can use file object in **for** loop (i.e., it is iterable)

```
f = open('somefile', 'r')
for line in f:
    if 'python' in line:
        print(line.strip())
```

File methods

- `read(size)` – Read and return *size* characters
- `readline()` – Read and return a single line
- `readlines()` – Return a list where each element is a line
- `seek(offset)` – Change position in file
- `tell()` – Return current position in file
- `flush()` – When writing, flush any buffers
- `close()` – Flush and close file

Exceptions

```
try:
    x = int(input("Enter a number: "))
except ValueError:
    print("Not a valid number!")
else:
    # No exceptions were raised
    print("Your number is " + str(x))
finally:
    # Always execute
    ...
```

Imports

Imports allow us to obtain variables defined in other files:

```
import decimal
import decimal as dc
from decimal import Decimal
from decimal import Decimal as Dec
```

Will discuss in-depth in week 6

Deleting names

- If we want to explicitly remove a variable that has been defined, we can use the **del** statement

```
>>> x = y = [1, 2, 3]
>>> del x
>>> y
[1, 2, 3]
>>> del y[1:]
>>> y
[1]
```

- Deleting a name only decreases the reference count; it doesn't actually delete the object!

Further details

Line continuation

Implicit line joining:

```
months = ['January', 'February', 'March', 'April',
          'May', 'June', 'July', 'August',
          'September', 'October', 'November', 'December']
```

Explicit line joining:

```
if 1900 < year < 2100 and 1 <= month <= 12 \
   and 1 <= day <= 31 and 0 <= hour < 24 \
   and 0 <= minute < 60 and 0 <= second < 60:    # valid date
       return 1
```

Mutable vs. Immutable

- An object is *immutable* if it cannot be changed in place after it has been created
- Immutable types in Python:
  - `int`, `float`, `complex`
  - `str`
  - `tuple`
  - `frozenset`

Equality vs. Identity

- == tells us if two objects are equal
- **is** tells us if two objects identities are the same

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a == b
True
>>> a is b
False
```

- **None** should always be compared with **is**, not ==

Bitwise operations on integers

Operation	Result
`x	y`	bitwise *or* of x and y
`x ^ y`	bitwise *exclusive or* of x and y
`x & y`	bitwise *and* of x and y
`x << n`	x shifted left by n bits
`x >> n`	x shifted right by n bits
`~x`	the bits of x inverted

String formatting

There are three ways to "format" strings:

- printf-style formatting
- str.format() method
- Formatted string literals, aka f-strings (Python 3.6)

Escape sequences

Escape Sequence	Meaning
`\\`	Backslash (\)
`\'`	Single quote (')
`\"`	Double quote (")
`\n`	ASCII linefeed (newline)
`\r`	ASCII carriage return
`\t`	ASCII horizontal tab
`\ooo`	Character with octal value *ooo*
`\xhh`	Character with hex value *hh*
`\N {name}`	Character named *name* in Unicode database
`\uxxxx`	Character with 16-bit hex value *xxxx*
`\Uxxxxxxxx`	Character with 32-bit hex value *xxxxxxxx*

Bytes objects

- As opposed to strings, which are sequences of Unicode characters, `bytes` objects are sequences of single bytes
- Create byte literals: `b'Some bytes'`

The `else` clause

Not well known, but you can use **`else`** in a **`while`** or **`for`** loop:

```
for x in seq:
    if x % 2 == 0:
        print('Sequence has an even number')
        break
else:
    print('Sequence does not have an even number')
```

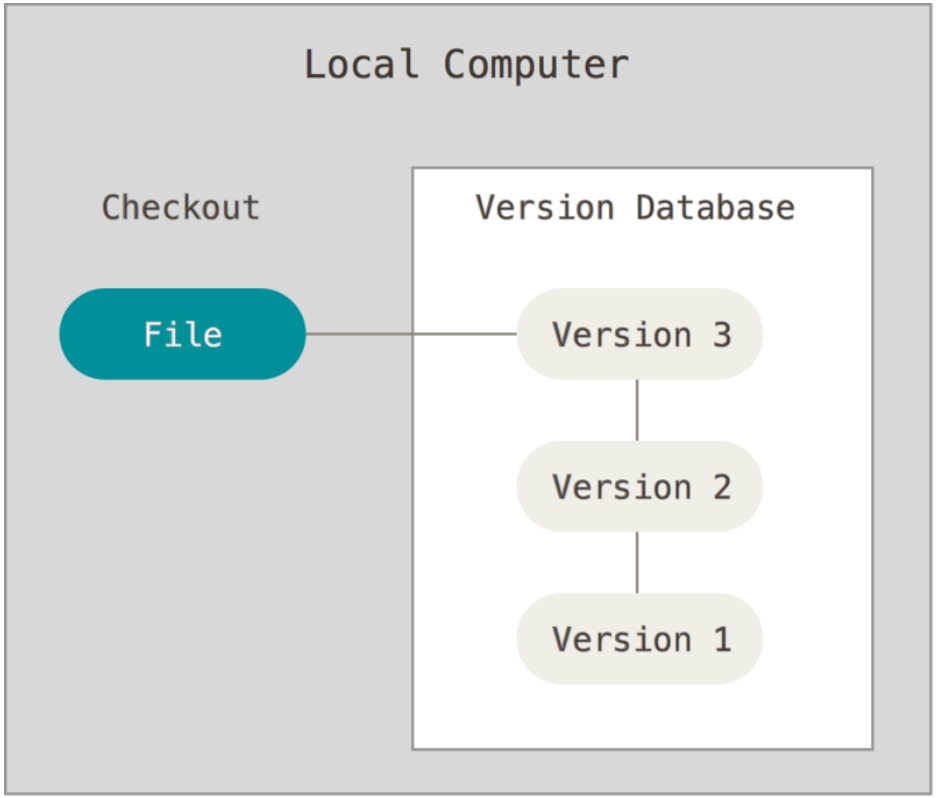
```
while foo:
    # Do something
else:
    # Execute once foo is False
```

git

git+GitHub

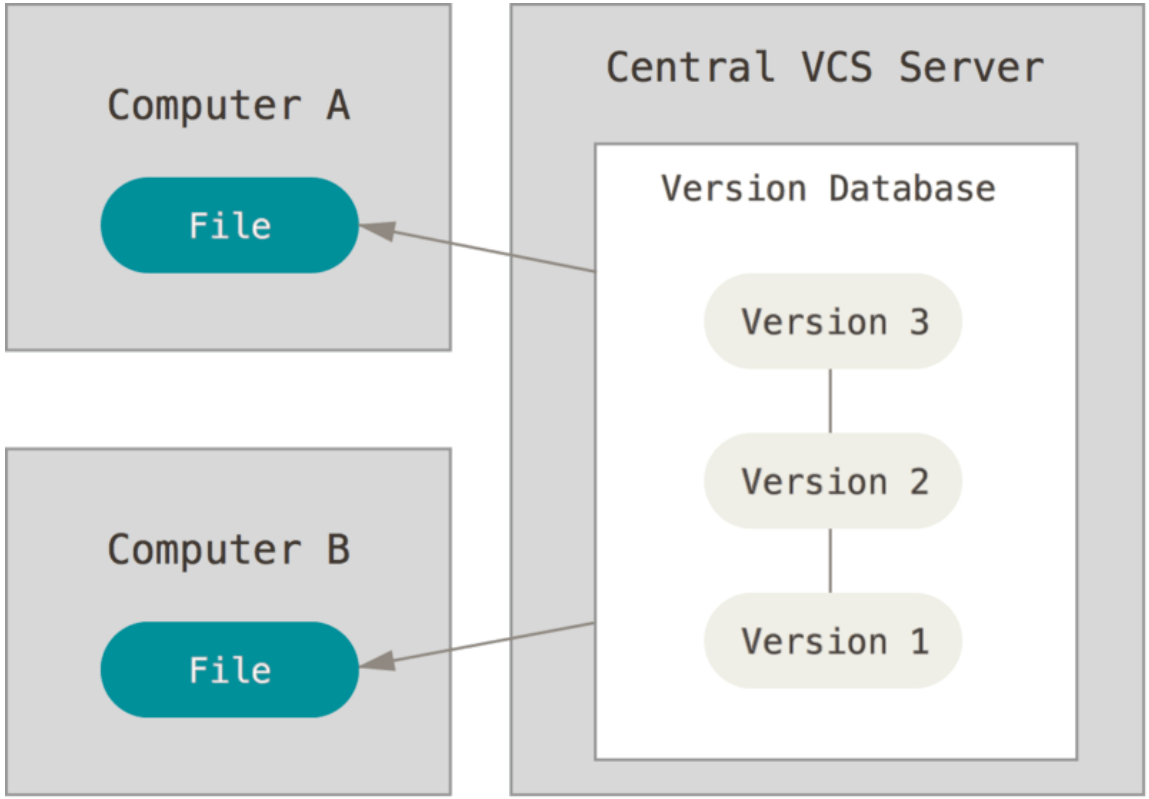
- We will be using GitHub for submitting assignments which relies on the git version control system
- Records changes to a file or set of files over time
  - Revert files to a previous state
  - Revert entire project to a previous state
  - Compare changes over time
  - See who last modified something and when
- Most commonly used for source code, but can (and often is) applied to other types of files

Local version control

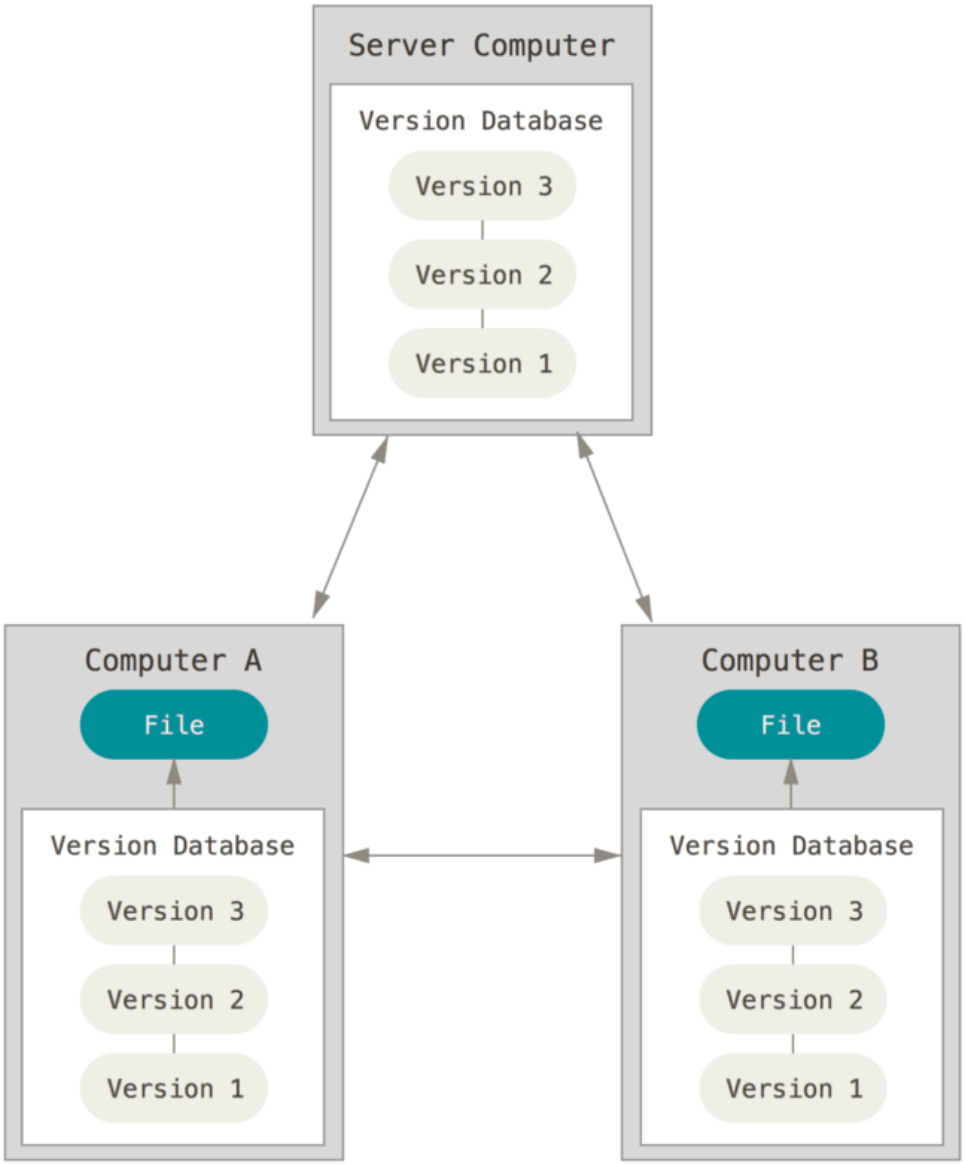


Source: Pro Git

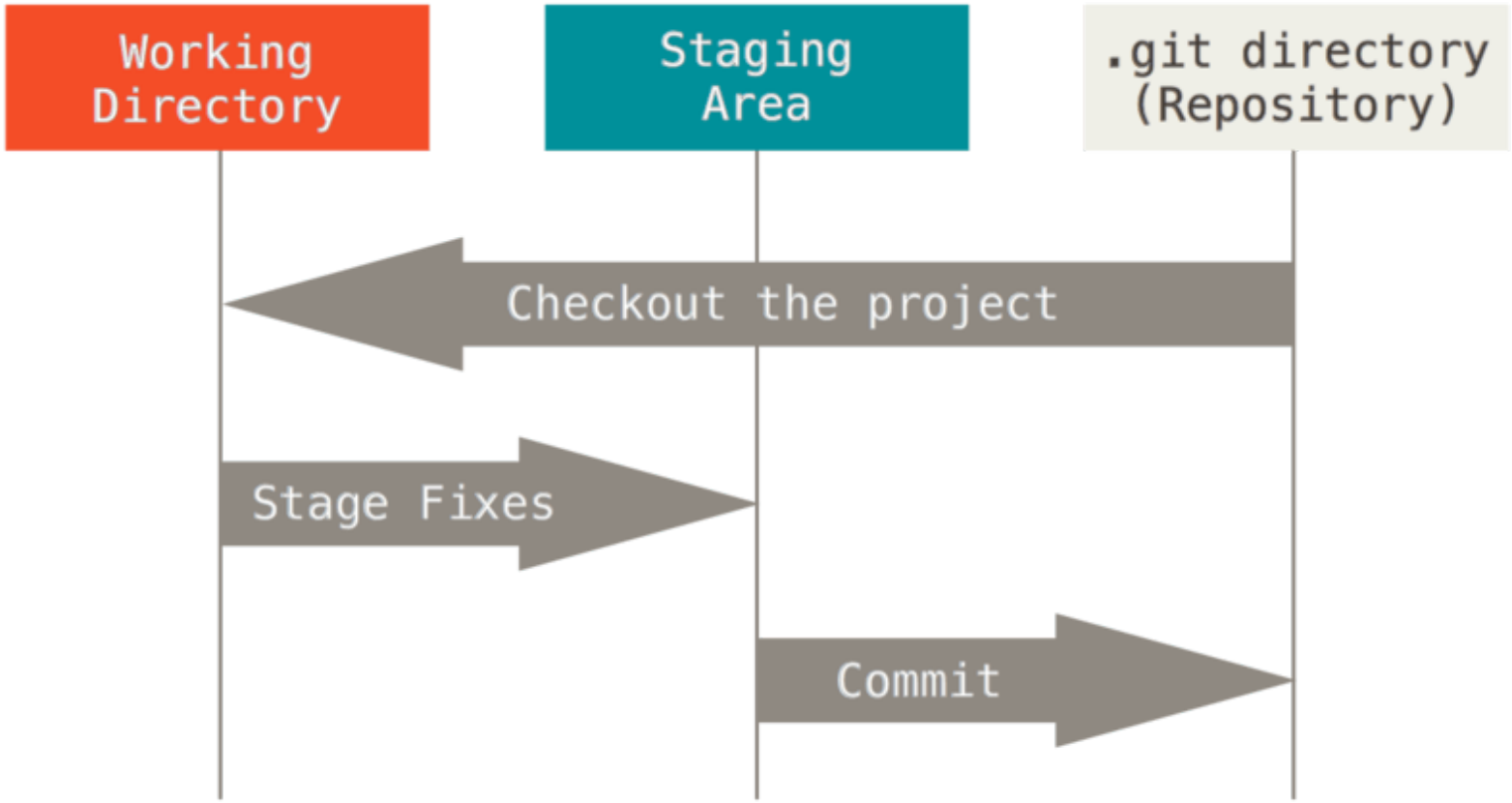
Centralized version control



Distributed version control



Local workflow



Primary commands

Command	Meaning	
`git init`	Initialize a new repository in current directory	local
`git add`	Stage new file or changes to a file	local
`git commit`	Commit changes to local repository	local
`git status`	See status of local repository	local
`git log`	See history of commits	local
`git merge`	Merge changes from remote repository	local
`git clone`	Clone a remote repository	remote
`git fetch`	Fetch changes from a remote repository	remote
`git pull`	Fetch changes and merge	remote
`git push`	Push changes on local branch to remote	remote

Assignment/Reading

Installing Python

- Plain vanilla Python: `https://www.python.org/`
- Easiest thing to do is to install a complete Python "distribution" that includes CPython + lots of third-party packages:
  - Anaconda
  - Miniconda

Installing on Windows

- Anaconda distribution provides you with most packages you'll need
- Other tools to consider:
  - cmder: command-line emulator (full version includes git)
  - GitHub Desktop

Installing on macOS

- /usr/bin/python on macOS is Python 2.7
- Anaconda distribution is recommended
- Other tools to consider:
  - Homebrew
  - MacPorts
  - GitHub Desktop

Installing on Linux

Using default Python

- Python 3 available at `/usr/bin/python3`

Packages can be installed:

- via package manager (yum, apt, etc.)
- Using `pip`
- Using `conda`

Text Editors/IDEs

- Sublime Text
- Notepad++ (Windows only)
- Atom
- Visual Studio Code
- PyCharm
- vim/emacs

Suggested Reading

- Chapter 1 in the git book
- Python Tutorial
- Learning Python: chapters 4, 6–9
- Python style guide: PEP8
- Start looking around Python documentation

Practice problems

Greatest common divisor

- MPCS placement, 2012-2013
- Recusive algorithm due to Dijkstra:

$$\gcd(a,b) = \begin{cases} a & \text{if } a = b \\ \gcd(a-b,b) & \text{if } a > b \\ \gcd(a,b-a) & \text{if } a < b \end{cases}$$

- Let's write a procedural version using **`while`**

Apaxian names

MPCS placement, 2013-2014