Lecture 2: Functions, Files, Collections, and Git

MPCS 51042-1 : Python Programming

Ron Rahaman

The University of Chicago, Dept of Computer Science

Oct 14, 2019

Table of Contents

Shared References (Lutz Ch. 6)

Functions (Lutz Ch. 10)

Files (Lutz Ch. 9)

More Collections

- Dictionaries (Lutz Ch. 9)
- Sets (https://docs.python.org/3.7/library/stdtypes.html#set-types-set-frozenset)
- Tuples (Lutz Ch. 9)
- Summarizing Data Structures

Doctests (https://docs.python.org/3.7/library/doctest.html)

Git (Chacon and Straub Ch. 2)

- Git Concepts
- Using Git

Table of Contents

Shared References (Lutz Ch. 6)

Functions (Lutz Ch. 10)

Files (Lutz Ch. 9)

More Collections

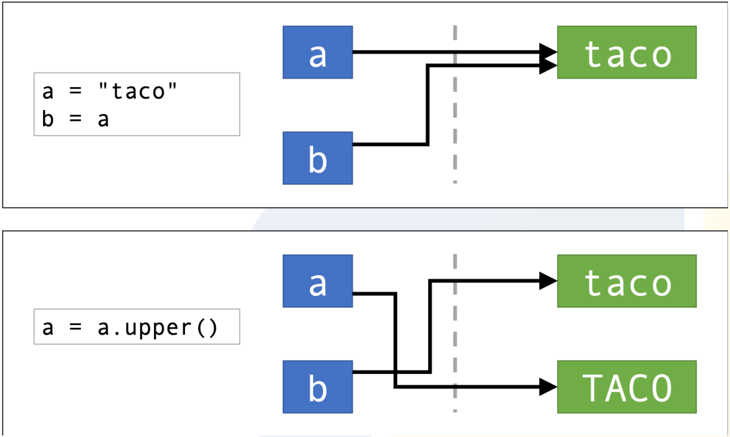
- Dictionaries (Lutz Ch. 9)
- Sets (https://docs.python.org/3.7/library/stdtypes.html#set-types-set-frozenset)
- Tuples (Lutz Ch. 9)
- Summarizing Data Structures

Doctests (https://docs.python.org/3.7/library/doctest.html)

Git (Chacon and Straub Ch. 2)

- Git Concepts
- Using Git

Shared References to Immutable Objects

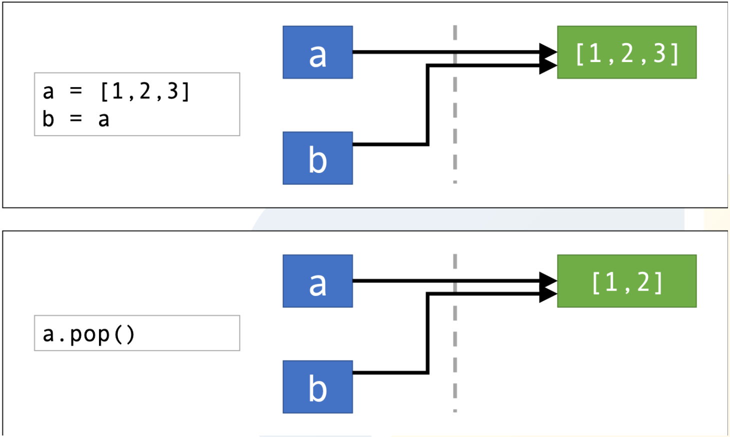


```
a = "taco"
b = a
```

```
a = a.upper()
```

- `a.upper()` creates a new string object. The name `a` now points to the new object.
- The name `b` still points to the original object.

Shared References to Mutable Objects



```
a = [1,2,3]
b = a
```

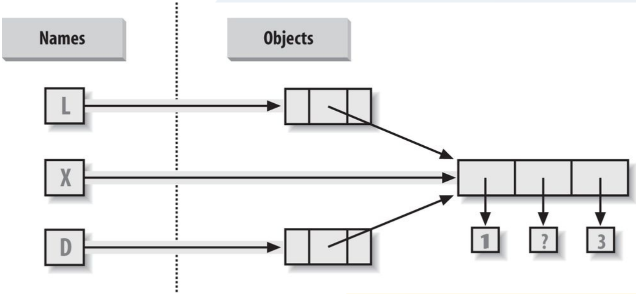
```
a.pop()
```

- a.pop() modifies the list object in-place. The name a still points to this object.
- The name b still points to the original object.

Shared References Inside Collections

These collections have shared references to the same list object.

```
X = [1, 2, 3]
L = ['a', X, 'b']
D = {'x':X, 'y':2}
```



What happens when we set `X[1] = 'surprise'`?

Table of Contents

Shared References (Lutz Ch. 6)

Functions (Lutz Ch. 10)

Files (Lutz Ch. 9)

More Collections

- Dictionaries (Lutz Ch. 9)
- Sets (https://docs.python.org/3.7/library/stdtypes.html#set-types-set-frozenset)
- Tuples (Lutz Ch. 9)
- Summarizing Data Structures

Doctests (https://docs.python.org/3.7/library/doctest.html)

Git (Chacon and Straub Ch. 2)

- Git Concepts
- Using Git

A First Example

```python
def intersect(seq1, seq2):
    res = []
    for x in seq1:
        if x in seq2:
            res.append(x)
    return res
```

- The function is not defined until the `def` statement is executed.
- When the `def` statement is executed, a new function object is created and bound to the name of the function.
- Arguments from caller are assigned to local variables.
- Variables defined inside function are also local.
- Returning a local variable removes the variable but keeps the object.

Thinking About Polymorphism

```python
def intersect(collect1, collect2):
    res = []
    for x in collect1:
        if x in collect2:
            res.append(x)
    return res
```

- Parameters types are not declared or checked.
- Type-checking could prevent future objects from working with this.
- In Python, we code for object interfaces, not object types.

Arguments and Shared References

In-place changes to shared objects can affect the caller.

```python
def changer(a, b):
    a = 2       # Changes local variables name
    b[0] = 2    # Changes shared object in-place


X = 1
L = ['a', 'b']
changer(X, L)  # L will be affected
```

Argument Matching from the Caller's Perspective

Given any function, the caller has several ways to match its arguments

- `func(value)` : Match argument by position
- `func(name=value)` : Match argument by keyword
- `func(*iterable)` : Unpack an iterable into positional arguments
- `func(**dict)` : Unpack a dict's key/value pairs into keyword args

These argument matching modes can be mixed, but must follow this order:

- Positional arguments
- Any combination of keyword and `*iter` args
- `**dict` args

Argument Matching in Function Definition

A function can define how its arguments are assigned to locals:

- `def f(name)` : Matched by position or keyword
- `def f(name=default)` : If not matched, define local to `default`
- `def f(*name)` : Matches and collects remaining positional arguments in tuple.
- `def f(**name)` : Matches and collects remaining keyword arguments in a dict.

These argument matching modes can be mixed, but must follow this order:

- Normal arguments
- Default arguments
- `*name` arguments
- `**dict` arguments

Argument Packing and Unpacking in Action

```python
def mymin(first, *rest):
    for arg in rest:
        if arg < first:
            first = arg
    return first
```

Many ways to call this!

Table of Contents

Shared References (Lutz Ch. 6)

Functions (Lutz Ch. 10)

Files (Lutz Ch. 9)

More Collections

- Dictionaries (Lutz Ch. 9)
- Sets (https://docs.python.org/3.7/library/stdtypes.html#set-types-set-frozenset)
- Tuples (Lutz Ch. 9)
- Summarizing Data Structures

Doctests (https://docs.python.org/3.7/library/doctest.html)

Git (Chacon and Straub Ch. 2)

- Git Concepts
- Using Git

Files

- A file object is instantiated by the `open()` built-in function
- The object's methods allow you to read/write strings .
- Useful methods include:
  - `f = open(filename, mode)` : Open a file and create a file object.
  - `s = f.read()` : Read the whole file into one string.
  - `s = f.readline()` : Read a single line in a file as a string.
  - `L = f.readlines()` : Read all lines into a list of strings.
  - `f.write(s)` : Write one string to a file.
  - `f.writelines(s)` : Write strings in a list to lines in a file.
  - `f.close()` : Close the file
- Can also pass an open file object to `print()`

File Modes

In `open(filename, mode)`, the mode is a string that specifies whether the file is read-only, write-only, etc.

- `r` : read-only
- `w` : write-only
- `a` : append
- `r+` : read-and-write

Files as Iterables and in Context Managers

When used as an iterator, file objects will return one line at a time:

```
f = open('myfile.txt', 'r'):
for line in f:
    # do something with line
f.close()
```

In a context manager, the file will be open and closed automatically:

```
with open('myfile.txt', 'r') as f:
    for line in f:
        # do something with line
```

File Examples

For these examples, use the "mpg" dataset from the seaborn packages (see `https://github.com/mwaskom/seaborn-data/blob/master/mpg.csv` and `http://archive.ics.uci.edu/ml/datasets/Auto+MPG`)

- Parse the .csv such that:
  - The header is in a list
  - The data are in a nested list
- Output the "name", "year", "weight", and "mpg" columns into a new file.

Table of Contents

Shared References (Lutz Ch. 6)

Functions (Lutz Ch. 10)

Files (Lutz Ch. 9)

More Collections

- Dictionaries (Lutz Ch. 9)
- Sets (https://docs.python.org/3.7/library/stdtypes.html#set-types-set-frozenset)
- Tuples (Lutz Ch. 9)
- Summarizing Data Structures

Doctests (https://docs.python.org/3.7/library/doctest.html)

Git (Chacon and Straub Ch. 2)

- Git Concepts
- Using Git

Table of Contents

Shared References (Lutz Ch. 6)

Functions (Lutz Ch. 10)

Files (Lutz Ch. 9)

More Collections

- Dictionaries (Lutz Ch. 9)
- Sets (https://docs.python.org/3.7/library/stdtypes.html#set-types-set-frozenset)
- Tuples (Lutz Ch. 9)
- Summarizing Data Structures

Doctests (https://docs.python.org/3.7/library/doctest.html)

Git (Chacon and Straub Ch. 2)

- Git Concepts
- Using Git

Dictionaries

Dictionaries are a mutable mapping type which associates keys with values

- The key must be a hashable type (integers, strings, tuples, etc.)
- The value is a reference and can refer to any object type.
- Keys are stored in a hash table for fast lookup. A particular ordering cannot be assumed.

Lots of ways to instantiate lists:

```
d = {'name': 'Harry', 'age': 3}      # {key1: val1, ...}

d = dict(name='Harry', age=3)        # dict(key1=val1, ...)

# dict(((key1,val1), (key2, val2), ...)
d = dict((('name', 'Harry'), ('age', 3)))

# dict(zip(key_list, val_list))
d = dict(zip(('name', 'age'), ('Harry', 3)))
```

Dictionary Operations

Dictionaries support operations of collection types but not sequences:

- d[k] returns an item. Raises KeyError if k is not in d.
- d[k] = v sets an item. If k is in d, the current value is replaced. If k is not in d, the new key:value pair is added to the dict.
- k in d returns True if the keys contain k.
- len(d) returns the number of keys

```
d = {}                  # An empty list
d['name'] = 'Harry'     # Now d is {'name': 'Harry'}
print(d['age'])         # This raises `KeyError`
'name' in d             # True
'Harry' in d            # False
```

Dictionary Views

In Python 3, several dict methods return views. Views are iterables that:

- Reflect future changes to the dictionary.
- Support set operations such as union and intersection.
- Are immutable.

The following methods return views:

- `d.items()`: A view of the dict's `(key, value)` pairs.
- `d.keys()`: A view of the dict's keys.
- `d.values()`: A view of the dict's values.

When used as an iterator, a dictionary object itself also returns an iterable over the keys:

```python
for key in d:
    print(key)
```

Getting Dict Items

Methods for getting items:

- `d.get(key[,default])`: If `key` is in dict, return its value. If not, return `default` if given or `None` if not given.
- `d.pop(key[,default])`: If `key` is in dict, return and remove its value. If not, return `default` if given or raise `KeyError` if not given.
- `d.popitem()`: v3.6 and before: remove and returns an arbitrary `(key, value)` pair. v3.7: remove a `(key, value)` pair in LIFO order.
- `d.copy()`: Retun a shallow copy of `d` Reference: `https://docs.python.org/3/library/stdtypes.html#mapping-types-dict`

Setting Dict Items

Methods for setting items:

- `d.setdefault(key[, default])`: If `key` is in dict, return its value. If not, add the `(key, default)` pair and return `default`.
- `d1.update(d2)`: Add the keys/value from `d2` to `d1`. Overwrites existing keys in `d1`.

The `collections.defaultdict` object provides similar functionality to `d.setdefault` but can be more convenient.

- `https://docs.python.org/3/library/stdtypes.html#mapping-types-dict`
- `https://docs.python.org/3.7/library/collections.html#collections.defaultdict`

Dict Examples

For these examples, use the "mpg" dataset from the seaborn packages (see `https://github.com/mwaskom/seaborn-data/blob/master/mpg.csv` and `http://archive.ics.uci.edu/ml/datasets/Auto+MPG`)

- Parse the .csv such that the entire table is a dict. The keys are column names and the values are a list of entries.
- Output the "name", "year", "weight", and "mpg" columns into a new file.

Table of Contents

Shared References (Lutz Ch. 6)

Functions (Lutz Ch. 10)

Files (Lutz Ch. 9)

More Collections

- Dictionaries (Lutz Ch. 9)
- Sets (https://docs.python.org/3.7/library/stdtypes.html#set-types-set-frozenset)
- Tuples (Lutz Ch. 9)
- Summarizing Data Structures

Doctests (https://docs.python.org/3.7/library/doctest.html)

Git (Chacon and Straub Ch. 2)

- Git Concepts
- Using Git

Sets

- Sets are unordered collections of unique, immutable, hashable objects.
- Sets may be instantiated by:
  - Literals: `s = {1, 2, 'a'}`
  - Function, using any iterable: `s = set([1, 2, 'a'])`
- Set operators accept only set objects:
  - Union: `s1 | s2`
  - Intersection: `s1 & s2`
  - Difference: `s1 - s2`
  - Symmetric difference: `s1 ^ s2`
  - Is subset: `s1 <= s2`
  - Is superset: `s1 >= s2`
  - etc.
- The corresponding set instance methods take any iterables:
  - `union()`, `intersection()`, `difference()`, `symmetric_difference()`, `issubset()`, `issuperset()`, etc.
- Reference: `https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset`

Set Demos

For these examples, consider the tests of “Frankenstein” and “Paradise Lost” from Project Gutenberg (`https://www.gutenberg.org/ebooks/84` and `https://www.gutenberg.org/ebooks/26`)

- Create sets of words from each text
- Compare the intersections of words from each text

Table of Contents

Shared References (Lutz Ch. 6)

Functions (Lutz Ch. 10)

Files (Lutz Ch. 9)

More Collections

- Dictionaries (Lutz Ch. 9)
- Sets (https://docs.python.org/3.7/library/stdtypes.html#set-types-set-frozenset)
- Tuples (Lutz Ch. 9)
- Summarizing Data Structures

Doctests (https://docs.python.org/3.7/library/doctest.html)

Git (Chacon and Straub Ch. 2)

- Git Concepts
- Using Git

Tuples

- Tuples are immutable, ordered collections of references.
  - After instantiated, their length is fixed.
  - Cannot be modified in-place.
- Sequence operations apply (indexing, slicing, concat).
- May be instantiated by:

```
t = ()               # An empty tuple
t = (1,)             # A single-element tuple
t = (1, 'a', [2, 3]) # Heterogenous, nested
t = 1, 'a', [2,3]    # Same as above
t = tuple(iterbl)    # From an iterable
```

- Immutability only applies to the reference, not the object itself.
  - E.g., this nested list is still mutable: `t=(1,'a',[2, 3])`
- Reference: https://docs.python.org/3/library/stdtypes.html#tuple

Frame Title

For these examples, use the “mpg” dataset from the seaborn packages (see `https://github.com/mwaskom/seaborn-data/blob/master/mpg.csv` and `http://archive.ics.uci.edu/ml/datasets/Auto+MPG`)

- Parse the .csv into the following data structure:
  - The table is a dict. Each key is the (model, year) of a car. The corresponding value is a list of the other data
- Parse the .csv into the following data structure:
  - The table is a dict. Each key is the (model, year) of a car. The corresponding value is another dict where the key is a field name; and the values are the value for that (model, year).

Table of Contents

Shared References (Lutz Ch. 6)

Functions (Lutz Ch. 10)

Files (Lutz Ch. 9)

More Collections

- Dictionaries (Lutz Ch. 9)
- Sets (https://docs.python.org/3.7/library/stdtypes.html#set-types-set-frozenset)
- Tuples (Lutz Ch. 9)
- Summarizing Data Structures

Doctests (https://docs.python.org/3.7/library/doctest.html)

Git (Chacon and Straub Ch. 2)

- Git Concepts
- Using Git

The Object Heirarchy: Everything is an object!

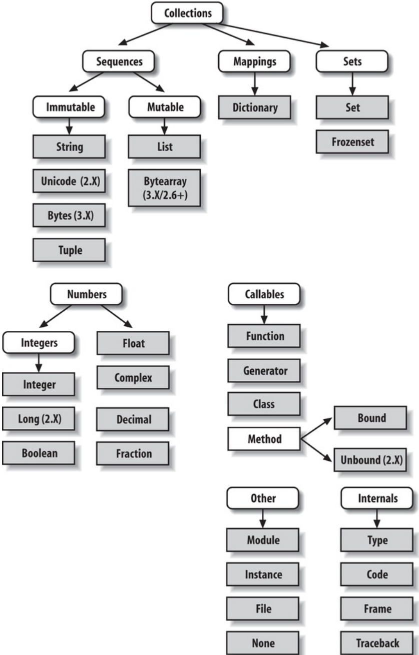


Table of Contents

Shared References (Lutz Ch. 6)

Functions (Lutz Ch. 10)

Files (Lutz Ch. 9)

More Collections

- Dictionaries (Lutz Ch. 9)
- Sets (https://docs.python.org/3.7/library/stdtypes.html#set-types-set-frozenset)
- Tuples (Lutz Ch. 9)
- Summarizing Data Structures

Doctests (https://docs.python.org/3.7/library/doctest.html)

Git (Chacon and Straub Ch. 2)

- Git Concepts
- Using Git

What are doctests?

- Doctests concisely provide both documentation and executable tests.
- Part of standard library: `https://docs.python.org/3.7/library/doctest.html`
- Compatable with many 3rd-party testing frameworks (pytest, nose)
- Can't deal with very complicated, multi-stage use cases. Need to use other frameworks, like unittest (`https://docs.python.org/3.7/library/unittest.html`)

Writing Doctests

- Written as a multiline string that shows an interactive Python session.
- Statements are run and compared to output.
- To pass, the output must match exactly.

The top of: out_of_class_demos/intersect.py

```
"""
>>> intersect([1, 2, 4], [6, 4, 2])
[2, 4]
>>> intersect([1, 2, 4], [4, 8, 2, 6])
[2, 4]
>>> intersect([1, 3, 5], [2, 4, 6])
[]
>>> intersect([1, 2, 4], [])
[]
>>> intersect("apple", "aerofoil")
['a', 'l', 'e']
"""
```

Running Doctests

- Can run on command line. Default usage only shows failed tests.

```
$ python3 -m doctest intersect.py
```

- Use the -v flag to show both passed and failed tests.

```
$ python3 -m doctest -v intersect.py
```

- Can run inside larger testing frameworks
- Can run inside PyCharm (maybe other IDEs?)

Choosing Tests

- Your program is only as good as its tests!
- Let's add this test and see what happens:

```
"""
>>> intersect('apple', 'pear')
['a', 'p', 'e']
"""
```

Table of Contents

Shared References (Lutz Ch. 6)

Functions (Lutz Ch. 10)

Files (Lutz Ch. 9)

More Collections

- Dictionaries (Lutz Ch. 9)
- Sets (https://docs.python.org/3.7/library/stdtypes.html#set-types-set-frozenset)
- Tuples (Lutz Ch. 9)
- Summarizing Data Structures

Doctests (https://docs.python.org/3.7/library/doctest.html)

Git (Chacon and Straub Ch. 2)

- Git Concepts
- Using Git

Table of Contents

Shared References (Lutz Ch. 6)

Functions (Lutz Ch. 10)

Files (Lutz Ch. 9)

More Collections

- Dictionaries (Lutz Ch. 9)
- Sets (https://docs.python.org/3.7/library/stdtypes.html#set-types-set-frozenset)
- Tuples (Lutz Ch. 9)
- Summarizing Data Structures

Doctests (https://docs.python.org/3.7/library/doctest.html)

Git (Chacon and Straub Ch. 2)

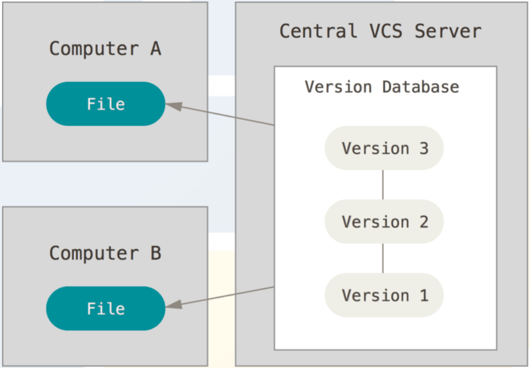
- Git Concepts
- Using Git

What is version control?

- A version control system (VCS) records changes to a set of files
  - All the recorded changes are called the history
  - The VCS can retrieve specific, previous versions of files
- Advantages of using VCS
  - If a bug is discovered after you deploy, you can revert to a working version
  - Can help discover which specific change caused the bug
  - Can discover which developer was responsible for that change

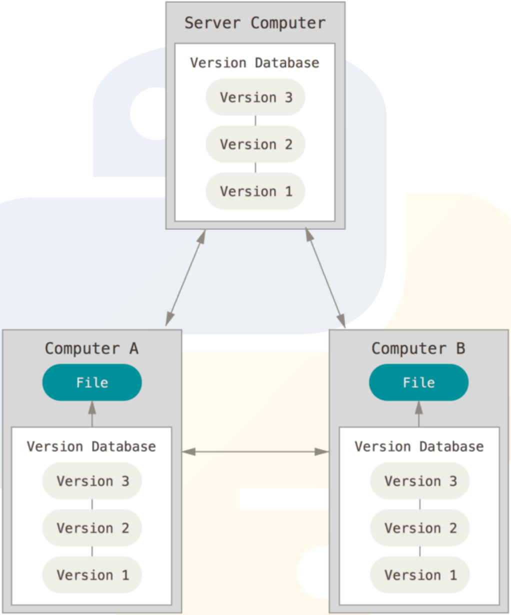
SVN etc.: Centralized VCS

- **Pros:** Individual devs can see everyone's activity. Admin has fine-grained control over database permissions.
- **Cons:** Single point of failure. Devs can't update database if their connection is offline.



Git: Distributed VCS

- Everyone has entire database:
  - Arbitrarily-many remote servers
  - Arbitrarily-many local clients
- **Pros:** Any server or client can used to recover the database. Clients can work offline. New non-linear workflows are possible.
- **Cons:** Learning curve.

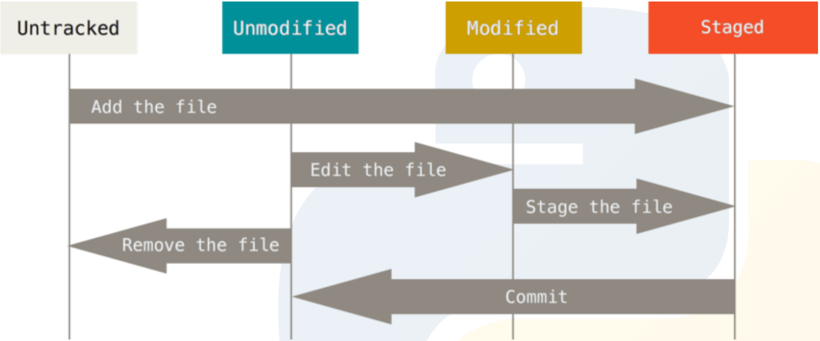


Areas in Your Local Project

```
drwxr-xr-x  12 rahaman  staff  384 Oct 10 18:02 .git
-rw-r--r--   1 rahaman  staff   88 Oct 10 18:02 .gitlab-ci.yml
-rw-r--r--   1 rahaman  staff  117 Oct 10 18:02 README.md
drwxr-xr-x   3 rahaman  staff   96 Oct 10 18:05 __pycache__
-rw-r--r--   1 rahaman  staff  637 Oct 10 18:03 my_abs.py
```

- **The Working Tree:** The files you're currently working on (my_abs.py, README.md, .gitlab-ci.yml)
- **The Git Directory:** The local database (.git)
- **The Staging Area (or Index):** Files are put here before updating the local database (not shown)

Lifecycle of Files



- Untracked: Not in local database
- Tracked: Some version in local database
  - Unmodified (or committed): Up-to-date with local database
  - Modified: File is changed in working
  - Staged: File's changes will go in next local database update

Table of Contents

Shared References (Lutz Ch. 6)

Functions (Lutz Ch. 10)

Files (Lutz Ch. 9)

More Collections

- Dictionaries (Lutz Ch. 9)
- Sets (https://docs.python.org/3.7/library/stdtypes.html#set-types-set-frozenset)
- Tuples (Lutz Ch. 9)
- Summarizing Data Structures

Doctests (https://docs.python.org/3.7/library/doctest.html)

Git (Chacon and Straub Ch. 2)

- Git Concepts
- Using Git

Getting and Setting-up Git

- Many choices of clients
  - We'll cover the command line tool (`https://git-scm.com/downloads`).
  - Many GUI clients are available (`https://git-scm.com/downloads/guis`).
  - Many IDEs have integrated clients (`https://www.jetbrains.com/help/pycharm/using-git-integration.html`)
- Pro Git Ch 1.6 describes first-time setup of command-line client (`https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup`)
  - Username and user email can be your CNET ID and @uchicago.edu email.
  - Editor can be anything
- To use UChicago's GitLab, you should set up an SSH key: `https://mit.cs.uchicago.edu/help/ssh/README.md`

Creating or Downloading Project

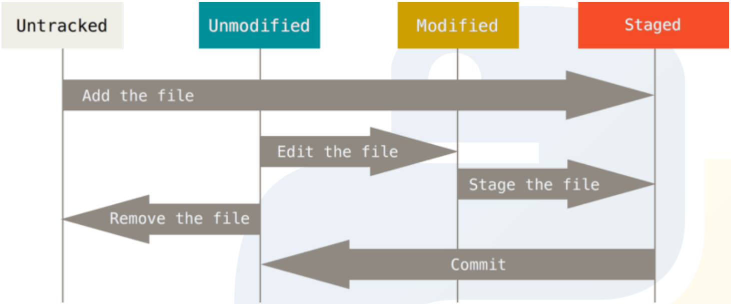
- We'll work with two demo repos in this lecture:
  - We'll be looking at this small pre-completed demo project: `https://mit.cs.uchicago.edu/mpcs51042-aut-19/git-demo-1`
  - We'll be developing this repo from scratch: `https://mit.cs.uchicago.edu/mpcs51042-aut-19/git-demo-2/`
- `git clone`: Download an existing repo

```
$ git clone git@mit.cs.uchicago.edu:mpcs51042-aut-19/git-demo-1.git
$ cd git-demo-1/
```

- `git init`: Create a new repo in the current directory

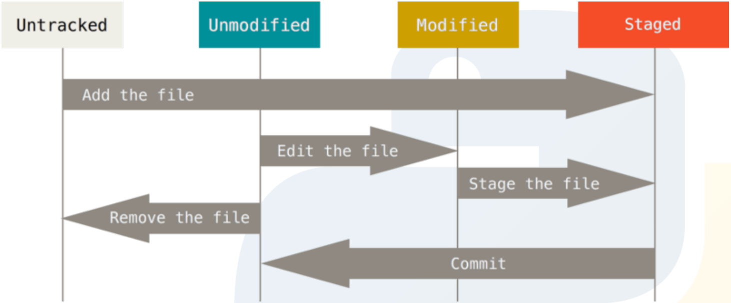
```
$ mkdir git-demo-2
$ cd git-demo-2/
$ git init
```

Adding Local Changes



- For new files:
  - **Untracked** → **Staged:** `git add <file>`
- For existing files:
  - **Unmodified** → **Modified:** Any text editor, IDE
  - **Modifed** → **Staged:** `git add <file>`
  - **Staged** → **Unmodified:** `git commit`

Undoing Local Changes



- **Staged → Modified:** `git reset -- <file>`
- **Modified → Unmodified:**
  - Lose changes forever: `git checkout -- <file>`
  - Keep changes for later: `git stash`
  - Apply stashed changes: `git stash apply`
- **Unmodified → Untracked:** `git rm --cached <file>`

Showing and Adding Remote Databases

- Every remote that is known to your local repo has:
  - A short name
  - A URL
- To show existing remotes:

```
$ git remote -v
```

- To add a new remote:

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

Getting Commits To and From Remotes

- To send committed, local changes to a remote

```
$ git push <remote_name> <branch_name>
```

- To get the latest changes from a remote

```
$ git pull <remote_name> <branch_name>
```

- By default, your repo will have
  - One remote: “origin”
  - One branch: “master”
- For homeworks, we will use multiple remotes. You won’t need to use multiple branches.