Computerlinguistische Anwendungen
Python/Git

Thang Vu

CIS, LMU
thangvu@cis.uni-muenchen.de

April 15, 2015

Introduction

- Review of Python
- Introduction to NumPy
- Introduction to Git

Core Data Types

Object type	**Example creation**
Numbers	123, 3.14
Strings	'this class is cool'
Lists	[1, 2, [1, 2]]
Dictionaries	{'1': 'abc', '2': 'def'}
Tuples	(1, 'Test', 2)
Files	open('file.txt'), open('file.bin', 'wb')
Sets	set('a', 'b', 'c')
Others	boolean, None
Program unit types	Functions, modules, classes

Variables

- store data, e.g., numbers
- content can be changed (is variable)
- have a data type
- assignment: var_name = value, e.g., `num = 17`

Dynamic Typing

- dynamic typing model
- types are determined automatically at runtime
- type of a variable can change
- check type with `type(var)`

Number Data Types

- integers, floating-point numbers, complex numbers, decimals, rationals
- Numbers support the basic mathematical operations, e.g.:
    - $+$ addition
    - $*$, $/$ multiplication, division
    - $**$ exponentiation
    - $<, >, <=, >=$ comparison
    - $! =, ==$ (in)equality

```
>>> 1/4
>>> float(1/4)
>>> float(1)/4
```

String Data Types

- immutable sequence of single characters
- **ASCII**: 256 characters: `'tree'`, `'2.1'`, `'two tokens'`
- **Unicode**: > 110,000 characters: `u'tree'`, `u'`σ`'`, `u'\u2B0000'`

Operations

```
s1 = ′the′
```

Operation	Description	Output
`len(s1)`	length of the string	3
`s1[0]`	indexing, 0-based	`′t′`
`s1[-1]`	backwards indexing	`′e′`
`s1[0:3]`	slicing, extracts a substring	`′the′`
`s1[:2]`	slicing, extracts a substring	`′th′`
`s1 + ′ sun′`	concatenation	`′the sun′`
`s1 * 3`	repetition	`′thethethe′`
! =, ==	(in)equality	True, False

String-Specific Methods

```
s1 = 'these'
```

Operation	Description	Output
`'-'.join(s1)`	concatenate with delimiter '-'	`'t-h-e-s-e'`
`s1.find('se')`	finds offsets of substrings	3
`s1.replace('ese', 'at')`	replace substrings, `s1` is still the initial string	`'that'`
`s1.split('s')`	splits string at delimiter	`['the', 'e']`
`s1.upper()`	upper case conversions	`'THESE'`
`s1.lower()`	lower case conversions	`'these'`

Lists

- collection of arbitrarily typed objects
- mutable
- positionally ordered
- no fixed size
- initialization: `L = [123, 'spam', 1.23]`
- empty list: `L = []`

Operations

```
L = [123, ’spam’, 1.23]
```

Operation	Description	Output
`len(L)`	length of the list	3
`L[1]`	indexing, 0-based	`’spam’`
`L[0:2]`	slicing, extracts a sublist	`[123, ’spam’, 1.23]`
`L + [4, 5, 6]`	concatenation	`[123, ’spam’, 1.23, 4, 5, 6]`
`L * 2`	repetition	`[123, ’spam’, 1.23, 123, ’spam’, 1.23]`

List-Specific Methods

```
L = [123, 'spam', 1.23]
```

Operation	Description	Output
`L.append('NI')`	append to the end	`[123, 'spam', 1.23, 'NI']`
`L.pop(2)`	delete item	`[123, 'spam']`
`L.insert(0, 'aa')`	insert item at index	`['aa', 123, 'spam', 1.23]`
`L.remove(123)`	remove given item	`['spam', 1.23]`
`L.sort()`	sort list	`[1.23, 123, 'spam']`
`L.reverse()`	reverse list	`[1.23, 'spam', 123]`

Nested Lists

Let us consider the 3*x*3 matrix of numbers `M = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`. *M* is a list of 3 objects, which are in turn lists as well and can be referred to as rows.

- `M[1]` – returns the second row in the main list: `[4, 5, 6]`
- `M[1][2]` – returns the third object situated in the in the second row of the main list: `6`

Dictionaries

- Dictionaries are not sequences, they are known as mappings
- They are mutable like lists
- They represent a collection of key-value pairs
- e.g.

```
>>> D = {'food':'Spam', 'quantity':4, 'color':'pink'}
```

Dictionary Operations

```
>>> D = {'food':'Spam', 'quantity':4, 'color':'pink'}
>>> D['food']           #Fetch value of key 'food'
'Spam'
>>> D['quantity'] += 1 #Add 1 to the value of 'quantity'
>>> D
D = {'food':'Spam', 'quantity':5, 'color':'pink'}
```

Dictionary Operations (cont.)

```
>>> D = {}
>>> D['name'] = 'Bob'     #Create keys by assignment
>>> D['job'] = 'researcher'
>>> D['age'] = 40

>>> D
D = {'name':'Bob', 'job':'researcher', 'age':40}

>>> print D['name']
Bob
```

Dictionary Operations (cont.)

```
>>> #Alternative construction techniques:
>>> D = dict(name='Bob', age=40)
>>> D = dict([('name', 'Bob'), ('age', 40)])
>>> D = dict(zip(['name', 'age'], ['Bob', 40]))
>>> D
{'age': 40, 'name': 'Bob'}
>>> #Check membership of a key
>>> 'age' in D
True
>>> D.keys()    #Get keys
['age', 'name']
>>> D.values() #Get values
[40, 'Bob']
>>> D.items()  #Get all keys and values
[('age', 40), 'name', 'Bob']
>>> len(D)      #Number of entries
2
```

Dictionary Operations (cont.)

```
>>> D = {'name': 'Bob'}
>>> D2 = {'age': 40, 'job': 'researcher'}
>>> D.update(D2)
>>> D
{'job': 'researcher', 'age': 40, 'name': 'Bob'}
>>> D.get('job')
'researcher'
>>> D.pop('age')
40
>>> D
{'job': 'researcher', 'name': 'Bob'}
```

Tuples

- Sequences like lists but immutable like strings
- Used to represent fixed collections of items

```
>>> T = (1, 2, 3, 4)   #A 4-item tuple
>>> len(T)             #Length
4
>>> T + (5, 6)         #Concatenation
(1, 2, 3, 4, 5, 6)
>>> T[0]               #Indexing, slicing and more
1
>>> len(T)
???
```

Sets

- Mutable
- Unordered collections of **unique** and **immutable objects**

```
>>> set([1, 2, 3, 4, 3])
set([1, 2, 3, 4])
>>> set('spaam')
set(['a', 'p', 's', 'm'])
>>> {1, 2, 3, 4}
set([1, 2, 3, 4])
>>> S = {'s', 'p', 'a', 'm'}
>>> S
set(['a', 'p', 's', 'm'])
>>> S.add('element')
>>> S
set(['a', 'p', 's', 'm', 'element'])
```

Files

- The main interface to access files on your computer
- Can be used to read and write text

```
>>> f = open('data.txt','w') #Make a new file in output mode 'w'
>>> f.write('Hello\n') #Write a string to it
4
>>> f.write('World\n')
(1, 2, 3, 4, 5, 6)
>>> f.close()          #Close to flush output puffers to disk
1
>>> #Continue writing at the end of an existing file
>>> f.write('data.txt', 'a')
>>> f.write('Cont.'\n)
>>> f.close()
```

Files

```
>>> f = open('data.txt')  #'r' is default processing mode
>>> text = f.read()       #Read entire file in a string
>>> text
Hello\nWorld\nCont.
>>> print text            #print interprets control characters
Hello
World
Cont.
>>> text.split()          #File content is always a string
['Hello', 'World', 'Cont.']
```

Larger files are always better read line by line!!!

```
>>> for line in open('data.txt','r'): print line
```

Immutable vs Mutable

Immutable:

- numbers
- strings
- tuples

Mutable:

- lists
- dictionaries
- sets
- newly coded objects

Testing: if statements

```
>>> x = 'killer rabbit'
>>> if x == 'roger':
...    print 'shave and a haircut'
... elif x == 'bugs':
...    print 'whats up?'
... else:
...    print 'run away!'
...
run away!
```

Note!

The `elif` statement is the equivalent of `else if` in Java or `elsif` in Perl.

Looping: while loops

```
>>> while True:
...       print 'Type Ctrl-C to stop me!'
...
>>> x == 'spam'
... while x:         #while x is not empty
...    print x
...    x = x[1:]
...
spam
pam
am
m
```

Looping: for loops

The for loop is a generic iterator in Python: it can step through the items in any ordered sequence or other iterable objects (strings, lists, tuples, and other built-in iterables, as well as new user-defined objects).

```
L = [1, 2, 3, 4]
for i in L:
    print i
```

```
for i in range(0, 5):
    print i
```

Looping: for loops

The most efficient file scanner in Python:

```
#use iterator: best for text input
>>> for line in open('data.txt'):
...      print line
```

Note!

This is not only the shortest but as well the most efficient coding for reading a file in Python. It relies on file iterators to automatically read one line on each loop iteration, which allows it to work with arbitrarily large files – often needed in NLP!

Function

- A function is a device that groups a set of statements so they can be run more than once in a program
- Why use functions?
  - Maximizing code reuse and minimizing redundancy
  - Procedural decomposition

Function: def statements

```
def name(arg1, arg2, ..., argN):
    statements
```

```
def name(arg1, arg2, ..., argN):
    ...
    return value
```

Function: def statements

```
def func(): ..    #Create function object
func()            #Call object
func.attr = value #Attach attributes
```

```
>>> def times(x, y):  #Create and assign function
...     return x*y    #Body executed when called
...
>>> times(2, 5)
10
```

Module

- Packaging of program code and data for reuse
- Provides self contained namespaces that minimize variable name clashes across programs
- The names that live in a module are called its attributes
- Typically correspond to Python program
- Module might be extensions coded in external languages such C++ or Java

Module

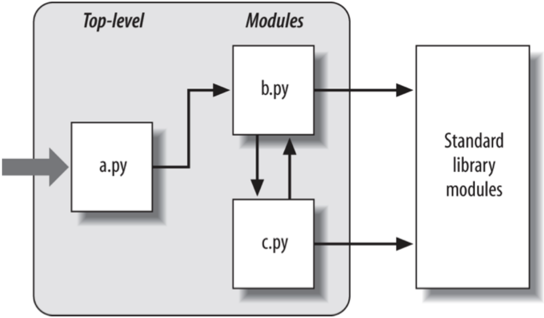
- `import` – Lets a client (importer) fetch a module as a whole
- `from` – Allows clients to fetch particular names from a module

Imports and Attributes

```python
# save in b.py
def spam(text):
    print text + ' spam'
```

```python
# File a.py
import b          #Import module b
b.spam('hallo')  #Print 'hallo spam'
```

Imports and Attributes



Regular Expressions

- Used to generate patterns that can be used to search for strings
- Is an algebraic formula whose value is a pattern consisting of a set of strings

regex		**string**
a	$\rightarrow$	a
ab	$\rightarrow$	ab
a$\star$	$\rightarrow$	a, aa, aaa, aaa . . .
a$\star$b$\star$	$\rightarrow$	ab, abb, aabb, aab . . .

Regular Expressions

- What can you match with the following regular expressions?

```
1. ^[Tt]the\b.*
2. [:;]-?[\|opPD\)\(]
3. <.*?>
4. \d+\-year\-old
```

Regular Expressions in Python

- To use Regular Expressions in Python, import the module `re`
- Then, there are two basic ways that you can use to match patterns:
  - `re.match()`
  - `re.search()`
- Both return `None` when there is no match

```python
import re

wordlist = ['farmhouse', 'greenhouse', 'guesthouse']

for w in wordlist:
     if re.match('(g.*?)(?=house)', w):
         print w
```

```python
match = re.search(pattern, string)
if match:
    process(match)
```

Regular Expressions in Python

Another way of using regular expressions is to compile them and reuse them as objects in your code.

```
import re

wordlist = ['farmhouse', 'greenhouse', 'guesthouse']
regex = re.compile('(g.*?)(?=house)')

for w in wordlist:
     if regex.match(w):
         print w
```

Python classes

```
class Classifier:
      def __init__(self, lambda1, lambda2):
          self.l1 = lambda1
          self.l2 = lambda2
      def train(self, data):
          ....
      def test(self, data):
          ....
if __name__ = '__main__':
     data = 'This is training data'
     testdata = 'This is test data'
     lambda1 = 0.002
                 lambda2 = 0.0005
                 model = Classifier(lambda1, lambda2)
     model.train(data)
     model.test(testdata)
```

- Access the data and the methods of each objects using `objectName.attributes` and `objectName.methods`

Storing objects

- **Objects** save data which we might want to reuse in the future
- Use `pickle`, you can save them and load them for reuse

import pickle

```python
import pickle

class Classifier:
      def __init__(self, params):
          self.params = params
      def setParams(self, params): ...
      def train(self, data): ....
      def test(self, testdata): ...

if __name__ = '__main__':
     params = [param1, param2, param3]
     data = 'This is training data'

     model = Classifier(params)
     model.train(data)
     #Store the model somewhere to reuse
     pickle.dump( model, open( 'model.p', 'wb' ) )
```

import `pickle`

```python
import pickle

class Classifier:
      def __init__(self, params):
          self.params = params
      def setParams(self, params): ...
      def train(self, data): ....
      def test(self, testdata): ....

if __name__ = '__main__':
     testdata = 'This is test data'

     model = pickle.load(open('model.p', 'rb'))
     model.test(testdata)
```

NumPy

- NumPy is a package supporting for **large, multi-dimensional arrays and matrices**, along with a large library of **high-level mathematical functions to operate on these arrays**
  - **ndarray** object is the core of the NumPy package
  - **ndarray** = n-dimensional arrays of homogeneous data
  - The standard mathematical and scientific packages in Python uses NumPy arrays
  - More information in http://www.numpy.org/
- NumPy will be helpful since machine learning works with high dimensional arrays

ndarray vs. list

- An ndarray is like a list
- However, there are several differences:
  - All the elements in an ndarray should have the same type. In a list, you can have different types
  - The number of elements in an ndarray is fixed, i.e. the number of elements cannot be changed
  - ndarray in NumPy is more efficient and faster than list

How to use NumPy

- Install NumPy (see HOWTO in http://www.scipy.org/scipylib/download.html)
- Take a look on the NumPy tutorial in www.scipy.org/Tentative_NumPy_Tutorial

NumPy: Should know

```
>>> import numpy as np
# Several ways to create a numpy array
>>> arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> a = np.ones((3,3), dtype=float)
>>> b = np.zeros((3,3), dtype=float)
>>> c = np.ones_like(arr)
>>> d = np.zeros_like(arr)
>>> e = np.identity(3, dtype = float)
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]])
```

NumPy: Should know

```
>>> import numpy as np
# create a numpy array from a list
>>> arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# returns the array dimension
>>> arr.ndim
2
# returns a tuple with the size of each array dimension
>>> arr.shape
(3, 3)
# returns the number of all the elements
>>> arr.size
9
# returns the transposed matrix
>>> arr.T
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
# returns the type of all the elements
>>> arr.dtype
dtype('int64')
```

NumPy: Should know

```
>>> import numpy as np
# create a numpy array from a list
>>> arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# array slicing and indexing
>>> arr[1]
array([4, 5, 6])
>>> arr[1][2]
6
>>> arr[:2]
array([1, 2, 3],
      [4, 5, 6])
>>> arr[1:]
array([4, 5, 6],
      [7, 8, 9])
>>> arr[1][:2]
array[4, 5]
```

NumPy: Should know

```
>>> import numpy as np
>>> arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
# array methods
>>> arr.flatten()
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = np.array([[1], [4], [7])
array([[1],
       [4],
       [7]])
>>> c = np.concatenate((arr, b), axis=1)
array([[1, 2, 3, 1],
       [4, 5, 6, 4],
       [7, 8, 9, 7]])
```

Version control system

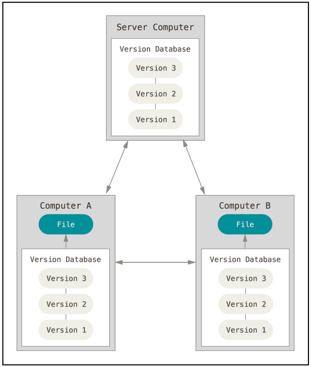
- >1 people work on the same documents (often code)
- It helps even if you are the only person who works on the code
  - Sometimes, you wish to go back to the past and undo your changes
- There are several VCS softwares such as svn, **git**, ...

Git - a distributed VCS

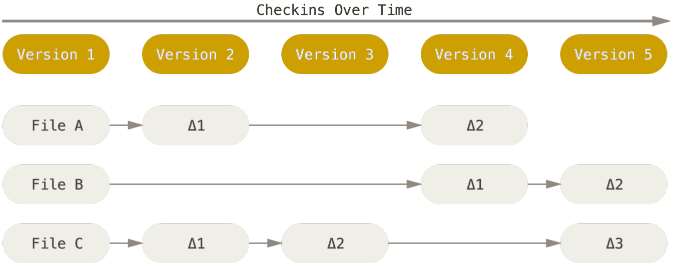
+ A free and open source distributed version control system

+ Is easy to use

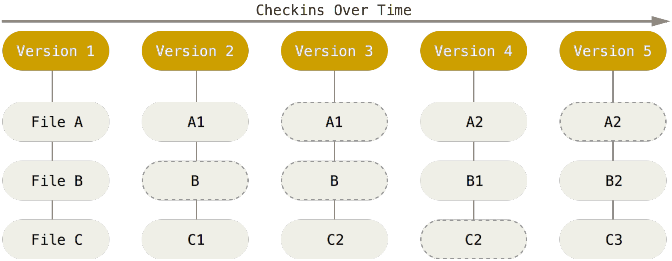
+ Very helpful in many contexts, especially in software development



Other subversion control systems



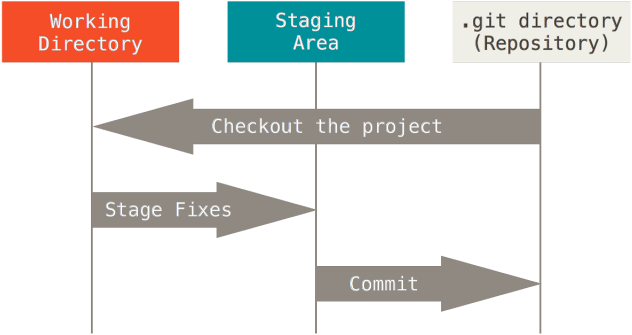
Git snapshot



Checkins Over Time

Version 1	Version 2	Version 3	Version 4	Version 5
File A	A1	A1	A2	A2
File B	B	B	B1	B2
File C	C1	C2	C2	C3

A Local Git Project



Git workflow

- Modify the files in local directory
- Stage the files and adding their snapshots in the staging area
- Do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to YOUR Git directory
- At the end, if everything is done, submit the changes to the repository on the server

How to start

- Get access to a Git server, e.g. CIP-Pool
  - Every student has a CIP-Pool access to the git server from IFI
  - You can request an account at: https://tools.rz.ifi.lmu.de/cipconf
- Install git on your machine
  - Generally you can find git at: http://git-scm.com/
  - For Unix Systems (Linux and MacOS): Most package managers have a git already as a package
  - For Windows: Windows users can get an easy installer at the above mentioned site

Starting a project

- There are two ways: **create** or **clone**
- Create a new project

```
$ git init [project_name]
# create new repository with specified name
# [project_name] can be omitted in make the working
    directory the git controlled folder
```

- Clone an existing project

```
$ git clone /path/to/repository
# clone a local repository
$ git clone [URL]
# clone remote repository
```

Make a change in your local

- Get status

```
$ git status
# Lists all new/modified files to be commited
```

- You can propose changes using:

```
$ git add [file]
# Snapshots the file in preparation for versioning
```

- Or sometime you want to remove them:

```
$ git reset [file]
# Unstages the file, but preserve its contents
```

- Then commit these changes

```
$ git commit -m "[descriptive message]"
# Records file snapshots permanently in version history
```

Submit the local change to the server

- With the command line:

```
$ git push [alias] [branch]
# Uploads all local branch commits
```

- Alias?
  - So that you don't have to use the full URL of a remote repository every time → Git stores an alias or nickname for each remote repository URL
  - By default, if you cloned the project (as opposed to creating a new one locally), Git will automatically add the URL of the repository that you cloned from under the name 'origin'
- Branch? At that point, you have only the 'master' branch
- Therefore, you will often use

```
$ git push origin master
```

Branch

- Branches are used to develop features isolated from each other
- The master branch is the "default" branch when you create a repository
- Use other branches for development and merge them back to the master branch upon completion

```
$ git branch
# Lists all local branches in the current repository
$ git branch [branch-name]
# Creates a new branch
$ git checkout [branch-name]
# Switches to the specified branch and updates the
    working directory
$ git merge [branch]
# Combines the specified branchs history into the
    current branch
$ git branch -d [branch-name]
# Deletes the specified branch
```

Branch

- However, if you changed the same part of the same file differently in the two branches, Git will not be able to merge them
- Git adds standard conflict-resolution markers to the files that have **conflicts**, so you can open them **manually and resolve those conflicts**

```
<<<<<<< HEAD:myfile
This line is written without newlines
=======
This line is written with a
newline
>>>>>>> [branch_name]:myfile
```

- To resolve this conflict you have to remove the annotation and keep one/rewrite the lines with the conflicts.

Update

- To get the latest updates you can use:

```
$ git pull
# Downloads bookmark history and incorporates changes
$ git diff
# Shows file differences not yet staged
$ git diff --staged
# Shows file differences between staging and the last file version
$ git fetch [bookmark]
# Downloads all history from the repository bookmark
```

Redo changes & Undo files

- Undo a commit

```
$ git reset [commit]
# Undoes all commits after [commit], preserving changes locally
```

- You should almost never do this

```
$ git reset --hard [commit]
# Discards all history and changes back to the specified commit
```

More about Git

- A compact list of all the important command lines on our website
- A good tutorial: http://gitimmersion.com/index.html
- Take the time and practice yourself

Q&A