From theory to practice: Standard tools Software carpentry, Part II

Pietro Berkes, Brandeis University

Outline

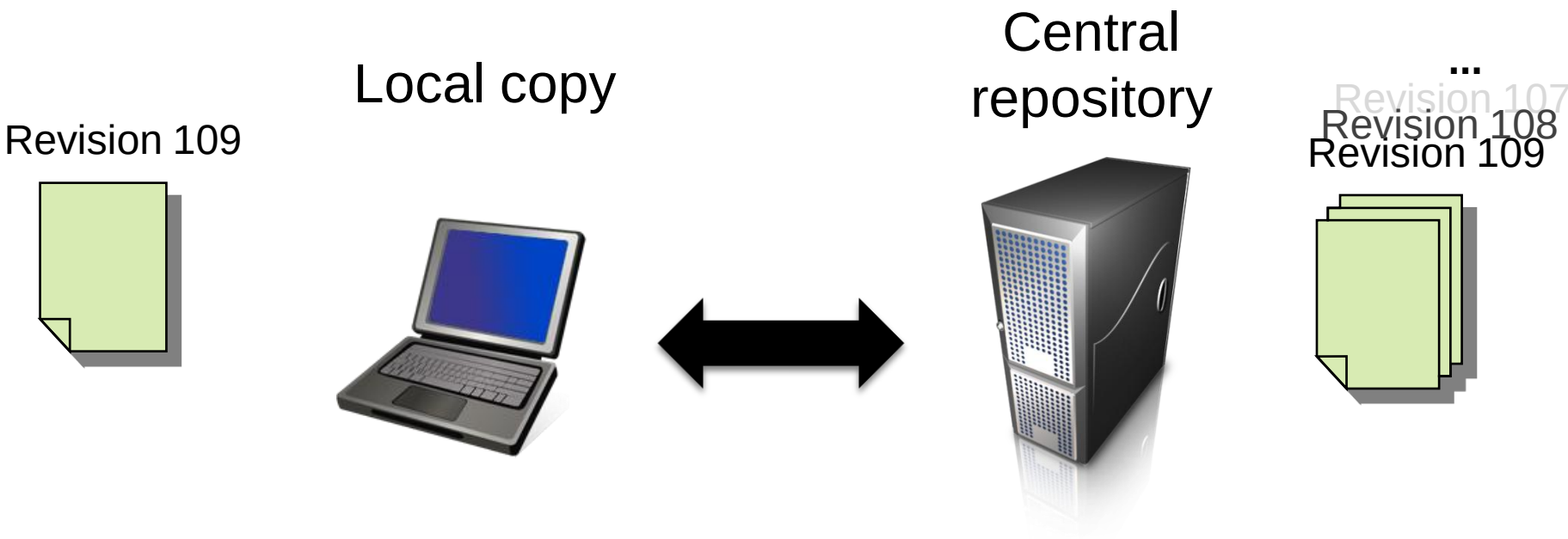
- Collaborating: SVN
- Profiling: `timeit, cProfile`
- Debugging: `pdb`
- Documentation, code clarity: `pydoc, pylint`

Python tools for agile programming

- I'll present:
  - Python standard "batteries included" tools
  - no graphical interface necessary
  - magic commands for ipython
- Many tools, based on command line or graphical interface
- Alternatives and cheat sheets are on the Wiki

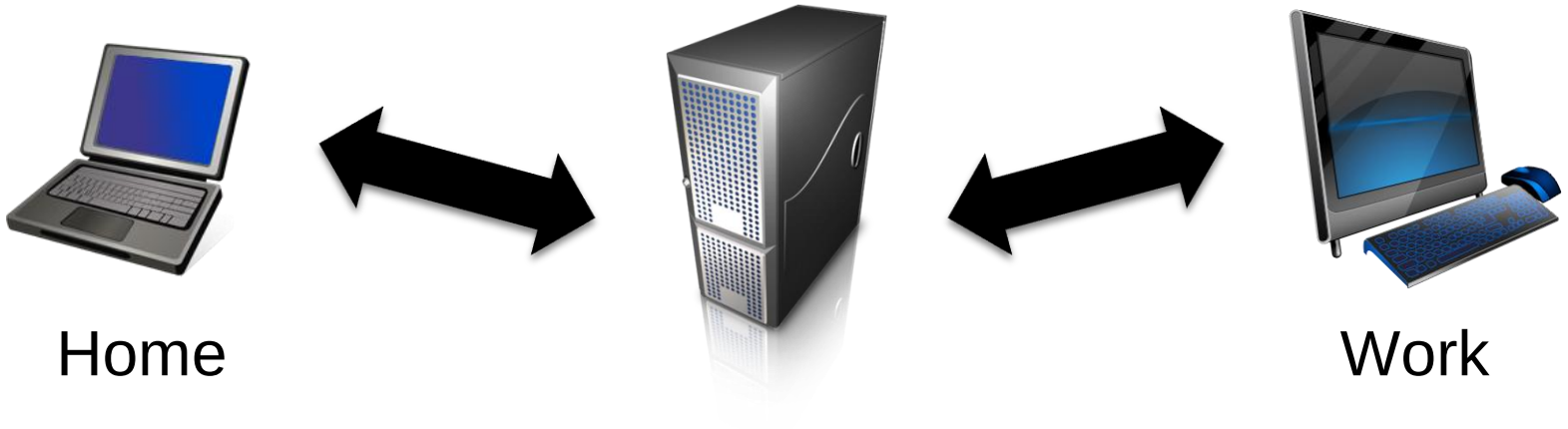
Version Control Systems

- Central repository of files and directories on a server
- The repository keeps track of changes in the files
- Manipulate versions (compare, revert, merge, …)



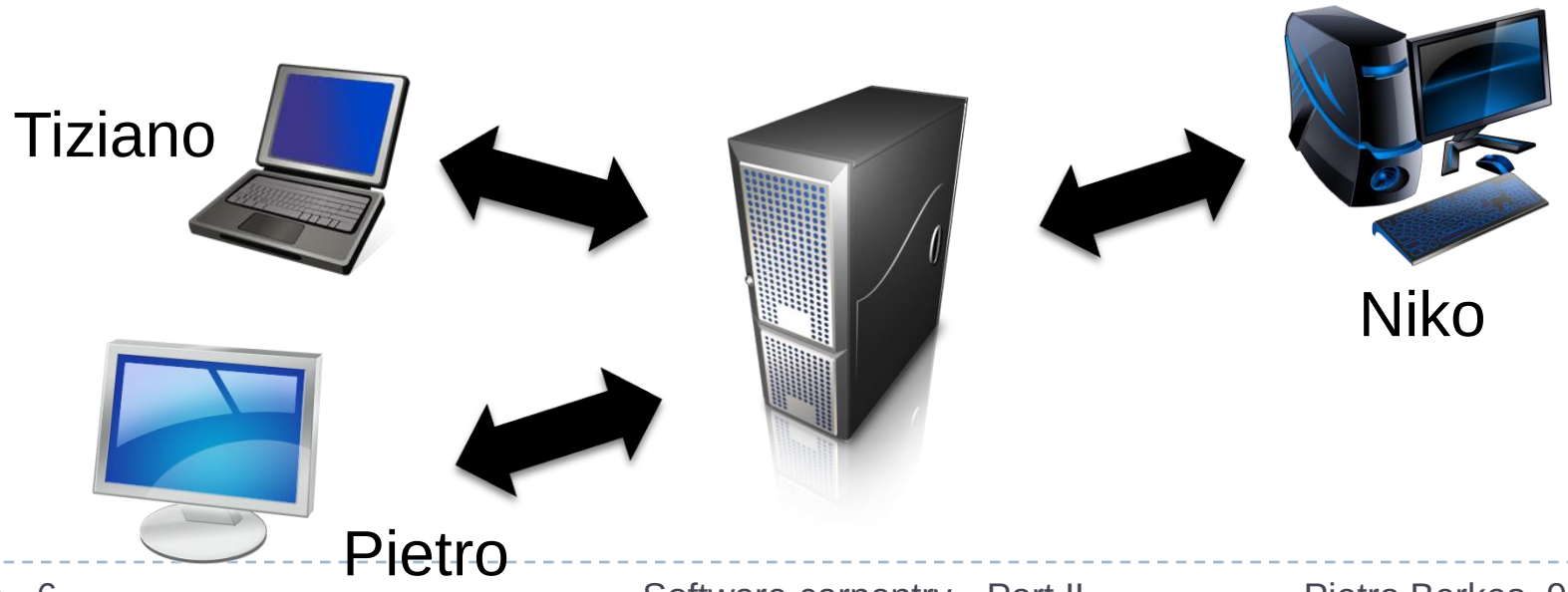
VCS for the lone scientist

- Store source code, data, papers, and presentations about a project
  - Backup
  - Reversible changes
  - Multiple synchronized copies of your project: now you can work from home, too!



VCS for a team of scientists

- Multiple people working at the same time on the same project (software libraries, papers)
  - Handle simultaneous changes to the same files and merge them or handle conflicts
  - Look at recent changes, who is responsible for newest versions, and much more



Subversion (SVN)

- Create a new repository

```
svnadmin create PATH
```

  - requires security decisions about access to repository, have a look at the SVN book

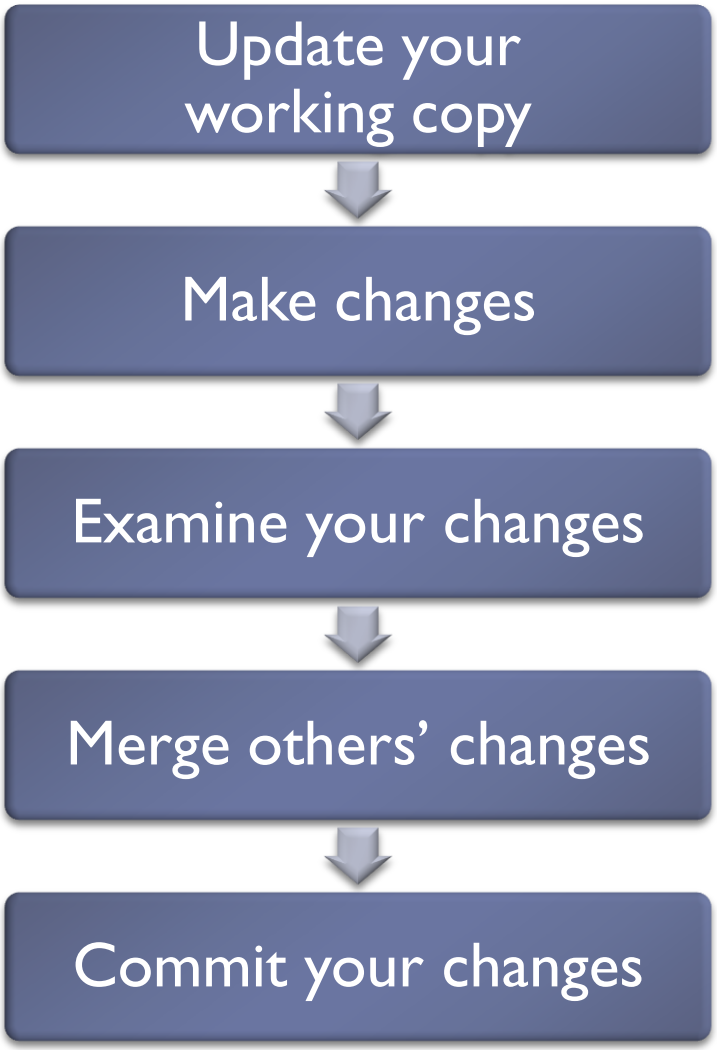
- Get a local copy of a repository

```
svn co URL [PATH]
```

- Checkout a copy of the course SVN repository

```
svn co https://portal.bccn-berlin.de/svn/python-summerschool/public
```

Basic SVN cycle




```
svn update
```

```
svn add       svn copy
svn delete    svn move
```

```
svn status    svn diff
svn revert
```

```
svn update
```
resolve conflicts then `svn resolved`

```
svn commit -m"meaningful message"
```


DEMO

SVN notes

- SVN cannot merge binary files => don't commit large binary files that change often (e.g., results files)
- At each milestone, commit the whole project with a clear message marking the event

```
svn commit –m"submission to Nature"
```

- There's more to it:
  - Branches, tags, repository administration
  - Graphical interfaces: subclipse for Eclipse, …
  - Distributed VCS: Mercurial, git, Bazaar

Test Suites in python: `unittest`

- Automated tests are a fundamental part of modern programming practices
- `unittest`: standard Python testing library

Anatomy of a TestCase

```
import unittest

class FirstTestCase(unittest.TestCase):

    def testtruisms(self):
        """All methods beginning with 'test' are executed"""
        self.assertTrue(True)
        self.assertFalse(False)

    def testequality(self):
        """Docstrings are printed during executions
        of the tests in the Eclipse IDE"""
        self.assertEqual(1, 1)

if __name__ == '__main__':
    unittest.main()
```

TestCase.assertSomething

```
assertTrue('Hi'.islower())              => fail
assertFalse('Hi'.islower())             => pass

assertEqual([2, 3], [2, 3])             => pass

assertAlmostEqual(1.125, 1.12, 2)       => pass
assertAlmostEqual(1.125, 1.12, 3)       => fail

assertRaises(exceptions.IOError, file,
             'inexistent', 'r')         => pass


assertTrue('Hi'.islower(),
           'One of the letters is not lowercase')
```

Multiple TestCases

```
import unittest

class FirstTestCase(unittest.TestCase):

    def testtruisms(self):
        self.assertTrue(True)
        self.assertFalse(False)

class SecondTestCase(unittest.TestCase):

    def testapproximation(self):
        self.assertAlmostEqual(1.1, 1.15, 1)

if __name__ == '__main__':
    # execute all TestCases in the module
    unittest.main()
```

setUp and tearDown

```
import unittest

class FirstTestCase(unittest.TestCase):

    def setUp(self):
        """setUp is called before every test"""
        pass

    def tearDown(self):
        """tearDown is called at the end of every test"""
        pass

    # ... all tests here ...

if __name__ == '__main__':
    unittest.main()
```


DEMO

Python code optimization

- Python is slower than C, but not prohibitively so
- In scientific applications, this difference is even less noticeable (`numpy`, `scipy`, ...)
  - for basic tasks, as fast as Matlab, sometimes faster
  - as Matlab, it can easily be extended with C or Fortran code
- Profiler = Tool that measures where the code spends time

timeit

- precise timing of a function/expression
- test different versions of a small amount of code, often used in interactive Python shell

```
from timeit import Timer

# execute 1 million times, return elapsed time(sec)
Timer("module.function(arg1, arg2)", "import module").timeit()

# more detailed control of timing
t = Timer("module.function(arg1, arg2)", "import module")
# make three measurements of timing, repeat 2 million times
t.repeat(3, 2000000)
```

- in ipython, you can use the `%timeit` magic command


DEMO

cProfile

- standard Python module to profile an entire application (`profile` is an old, slow profiling module)
- Running the profiler from command line:

```
python -m cProfile myscript.py
```

options `-o output_file`

`-s sort_mode` (`calls`, `cumulative`, `name`, ...)

- from interactive shell/code:

```
import cProfile
cProfile.run(expression[,"filename.profile"])
```

`cProfile`, analyzing profiling results

- From interactive shell/code:

```
import pstat
p = pstat.Stats("filename.profile")
p.sort_stats(sort_order)
p.print_stats()
```

- Simple graphical description with RunSnakeRun


DEMO

`cProfile`, analyzing profiling results

- Look for a small number of functions that consume most of the time, those are the *only* parts that you should optimize
- High number of calls per functions
  => bad algorithm?
- High time per call
  => consider caching
- High times, but valid
  => consider using libraries like `numpy` or rewriting in C

Debugging

- The best way to debug is to avoid it
- Your test cases should already exclude a big portion of the possible causes
- Don't start littering your code with *print* statements
- Core idea in debugging: you can stop the execution of your application at the bug, look at the state of the variables, and execute the code step by step

`pdb`, the Python debugger

- Command-line based debugger
- `pdb` opens an interactive shell, in which one can interact with the code
  - examine and change value of variables
  - execute code line by line
  - set up breakpoints
  - examine calls stack

Entering the debugger

- Enter at the start of a program, from command line:

```
python -m pdb mycode.py
```

- Enter in a statement or function:

```
import pdb
# your code here
if __name__ == '__main__':
    pdb.runcall(function[, argument, ...])
    pdb.run(expression)
```

- Enter at a specific point in the code:

```
import pdb
# some code here
# the debugger starts here
pdb.set_trace()
# rest of the code
```


DEMO

Entering the debugger

- From ipython:
  `%pdb` - preventive
  `%debug` – post-mortem

Two more useful tools

- `pydoc`: creating documentation from your docstrings

```
pydoc [-w] module_name
```

- `pylint`: check that your code respects standards


DEMO

The End

- Exercises after the tea break...

		1						
		2		3				4
			5			6		7
5			1	4				
	7						2	
				7	8			9
8		7			9			
4				6		3		
						5		

TestCase.assertSomething

TestCase methods	Examples

`assert_(expr[, msg)`
`assertTrue(expr[, msg])`
`assertFalse(expr[, msg])`
	`assertTrue(isinstance([1,2], list) => pass`
	`assertTrue('Hi'.islower()) => fail`

`assertEqual(first, second[, msg])` `assertNotEqual(first, second[, msg])`	`assertEqual([2, 3], [2, 3]) => pass` `assertEqual(1.2, 1.3) => fail`
`assertAlmostEqual(first, second [, places[, msg]])` `assertNotAlmostEqual(first, second [, places[, msg]])`	`assertAlmostEqual(1.125, 1.12, 2) => pass` `assertAlmostEqual(1.125, 1.12, 3) => fail`
`assertRaises(exception, callable, ...)`	`assertRaises(exceptions.IOError, file, 'inexistent', 'r') => pass` `assertRaises(exceptions.SyntaxError, file, 'inexistent', 'r') => fail`
`fail([msg])`	`fail() => fail`