NPRG065: Programming in Python
Lecture 6

http://d3s.mff.cuni.cz

Department of Distributed and Dependable Systems

Tomas Bures

Petr Hnetynka

{bures,hnetynka}@d3s.mff.cuni.cz

CHARLES UNIVERSITY IN PRAGUE

faculty of mathematics and physics

Std library overview
(Important modules)

Department of
Distributed and
Dependable
Systems

Logging

- `import logging`
- Similar to any other logging framework
- 5 levels
  - DEBUG, INFO, WARNING, ERROR, CRITICAL
- Loggers
  - hierarchical names
- Logging configuration – handlers, formatters
  - in code
  - external file
    - several formats

See
logs/*.py

Low level OS functions

- `import os`
- Operating system API

See
os/os.py – Miscellaneous operating system API
os/os.file.py – File operating system API

General – different file access APIs

- There are several ways how to access files in Python
  - Build-in open()
    - This is a generic way how to open files.
    - Use this if there are no special requirements to use os API.
    - Returns a file object with read, write, … methods.
  - pathlib Path.open()
    - Behaves like open() but provides nice path abstraction.
    - Returns the same file object.
  - os.open()
    - Provides low level file API, maps to native C functions.
    - Returns native file descriptor as used by the underlying operating system (an integer).
    - os contains methods for low level file access
      - File is passed in form of a file descriptor
      - Some methods also accept file name if possible
        - For instance os.lseek does not make sense with just file name
    - Use when necessary

os – low level file access API

- There used to be 2 versions of each function
  - One for working with path (like os.stat)
  - Another one for working with file descriptors (like os.fstat)
  - Since Python 3.3 the os.stat and similar methods naturally working with paths also take fd or dir_fd argument, thus the fd only versions prefixed with f are redundant.
- Everything does not work everywhere
  - Quite big part of the API is Unix only.
  - Sometimes only part of the functionality is available.
  - Sometimes the result of the operation is platform dependent.
  - It is possible to ask whenever particular function supports something by checking the function being present in os.supports_...
    - os.supports_dir_fd
    - os.supports_effective_ids
    - os.supports_fd
    - os.supports_follow_symlinks

The os file API is similar to C file API

Windows, Unix, usually Mac		Unix only
os.open	os.mkfifo	os.chown
os.close	os.readlink	os.get_blocking
os.dup	os.remove	os.lockf
os.pipe	os.removedirs	os.possix_fallocate
os.read	os.rename	os.possix_fadvise
os.sendfile	os.replace	os.set_blocking
os.write	os.rmdir	os.chroot
os.access	os.scandir	os.sync
os.chdir	os.stat	
os.chflags	os.stat_float_times	
os.chmod	os.symlink	
os.getcwd	os.truncate	
os.link	os.unlink	
os.listdir	os.utime	
os.lstat	os.walk	
os.mkdir	os.mkdirs	

File path access via pathlib

- `import pathlib`
- Working with filesystem paths

See
path.py

Argument parsing

- `import argparse`
- Parsing command-line arguments

See
arguments.py

Regular expressions

- `import re`
- Regular expression support

See
regexp.py

System

- `import sys`
- System-specific parameters and functions

See
system.py

Shell utils

- **`import shutil`**
- High-level file API

See
sh/sh.py

XML

- **`import xml`**
- Parsing XML documents

See
xml/xmltree.py

CSV

- `import csv`
- Reading and writing CSV files

See
csv/*.py

JSON

- **`import json`**
- Reading and writing json formatted data

YAML

- Is not in the std library
  - **`pip install pyyaml`**
  - Or better **`pip install ruamel.yaml`**
    - Newer, maintained, supports yaml 1.2
    - Almost the same usage (API)

- **`import yaml`**
- Reading and writing yaml formatted data

See
y/y.py

Unit testing

Unit testing

- unit testing
  - testing "small" units of functionality
  - a unit – independent on other ones
  - tests are separated
    - creating helper objects for tests
      - context
  - typically in OO languages
    - unit ~ method
  - ideally – unit tests for all units in a program
    - typically in OO languages
      - for all public methods

Unit testing in Python

- Modules in std. library
  - doctest
  - unittest

doctest

- Placing testing code in pydoc comments

```
def echo(value):
    """
    Documentation here

    >>> echo(0)
    0
    """
    return value
```

Code to be executed as a test

Expected value of the test

- Executing tests
  - python -m doctest -v example.py
- Or
  - placing doctest.testmod() to “main” and executing the module with the argument -v

See doctesting.py

unittest

- Tests in a special class

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

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

Have to extend this class

Individual tests

Many assertSomething methods for evaluation conditions. If the condition is true, the assertSomething method does nothing. If not true, an exception is raised, i.e., the test fails.

unittest

Method	Checks that
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)

Method	Checks that
assertAlmostEqual(a, b)	round(a-b, 7) == 0
assertNotAlmostEqual(a, b)	round(a-b, 7) != 0
assertGreater(a, b)	a > b
assertGreaterEqual(a, b)	a >= b
assertLess(a, b)	a < b
assertLessEqual(a, b)	a <= b
assertRegex(s, r)	r.search(s)
assertNotRegex(s, r)	not r.search(s)
assertCountEqual(a, b)	*a* and *b* have the same elements in the same number, regardless of their order.

Method	Checks that
assertRaises(exc, fun, *args, **kwds)	fun(*args, **kwds) raises *exc*
assertRaisesRegex(exc, r, fun, *args, **kwds)	fun(*args, **kwds) raises *exc* and the message matches regex *r*
assertWarns(warn, fun, *args, **kwds)	fun(*args, **kwds) raises *warn*
assertWarnsRegex(warn, r, fun, *args, **kwds)	fun(*args, **kwds) raises *warn* and the message matches regex *r*
assertLogs(logger, level)	The with block logs on *logger* with minimum *level*

unittest

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                                 'incorrect default size')

    def test_widget_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(),
              (100,150), 'wrong size after resize')

    def tearDown(self):
        self.widget.dispose()
```

Called before each test method

Called after each test method

unittest

- Methods called before/after each all tests in a particular class

```
@classmethod
def setUpClass(cls):
    ...

@classmethod
def tearDownClass(cls):
    ...
```

- Tests execution
  - python -m unittest test_module1 test_module2

See unittesting.py

Packing and distributing code

Installing packages using PIP

- PIP – a tool that enables automated installation of packages from a large repository
  - packages from pypi.org
- As of Python 3.4 PIP is part of the default Python installation
- Usage:
  - python -m pip install SomePackage
  - python -m pip install –user SomePackage
  - python -m pip install SomePackage==1.0.4
  - python -m pip install --upgrade SomePackage
- Problems:
  - May interfere with system package managers on Posix systems
    - install package just for single user using “--user” or use virtual environment
      - described later
  - Packages with native content need to be build from source

Department of Distributed and Dependable Systems

Installing packages from source

- By convention installable Python sources have setup.py installation script in their root directory
- setup.py should ensure installation of the packages and modules included in the codebase as intended by author.
- It can be invoked as this:
  - python setup.py install
  - python setup.py install –user

    - if possible, prefer PIP and pypi.org

Department of Distributed and Dependable Systems

Virtual Environment

- venv
  - a tool for creating virtual Python environments

```
python3 -m venv DIRECTORY
```

  - sets up virtual environment in the DIRECTORY
    - new packages are installed to the DIRECTORY

```
source /path/to/DIRECTORY/bin/activate
```

  - activates the environment
- virtualenv
  - similar, just another package for the same

```
python3 -m virtualenv DIRECTORY
```

Managing Dependencies

- pipenv
  - combination of PIP and virtualenv
- creates virtualenv and install dependencies there
- list of dependencies stored in a file within the project

```
cd myproject
pipenv install <package>
pipenv shell
```

Packaging Applications

- setuptools

- Tool for packaging python applications
- … and describing requirements

- Driven by setup.py

Department of Distributed and Dependable Systems

Writing setup.py

- In theory any arbitrary code can be in setup.py
  - it is a normal script
  - but typically contains only the package description
- In fact all the installation code does not need to be written again
  - The setuptools package contains the necessary functions
  - Particularly the setup function is used to configure what to install
  - For most projects a call to the setup is everything that is needed

See
myhello directory
and setup.py there

What does an installed package look like

- Packages are installed as python eggs
  - each installed package has a directory or an egg archive containing its files:
    - python source code
    - any other resource necessary for the package to work properly
    - precompiled .pyc files in the __pycache__ subdirectory
  - each package also has its own text file describing package metadata
    - contains name, version, summary, url, authors, licence, dependencies, …

Where are the installed packages

- Python looks for packages to import on several places
- The lookup is controlled by the Python Path variable
- By default it contains:
  - the directory where the script is located
  - python installation package directory
  - other system Python packages (site-packages directory)
  - user local package directory
  - content of PYTHONPATH environment variable
- Path can be accessed and modified at runtime
  - import sys
  - print(sys.path)
  - sys.path.append("some path")

Department of Distributed and Dependable Systems