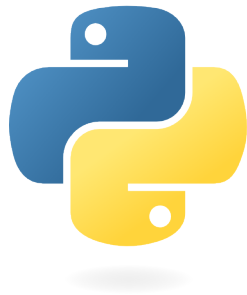


Programming in Python – Exercises



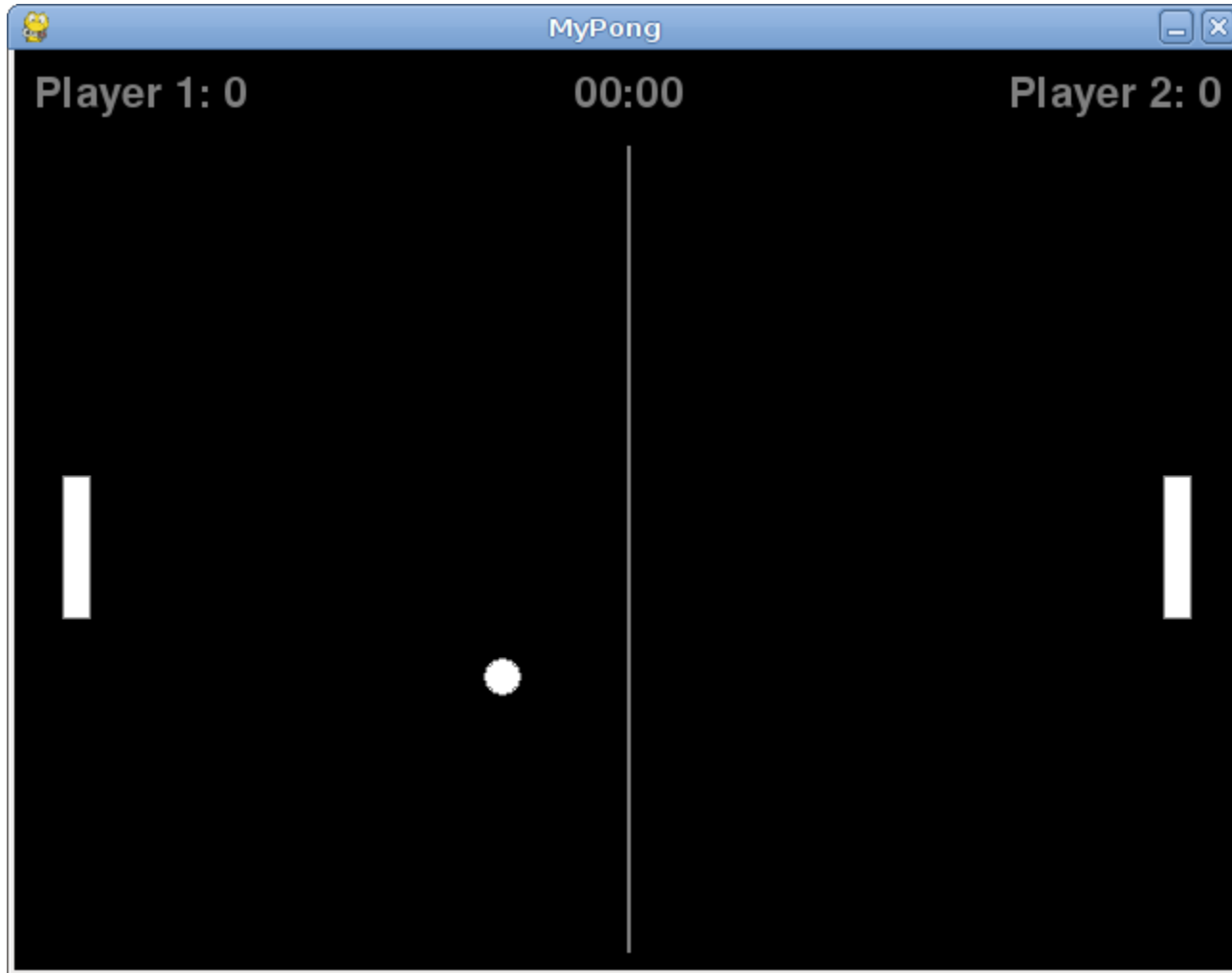
PyWorkshop - June 09 2008

Jochen Martin Eppler

Honda Research Institute Europe GmbH
63073 Offenbach/Main, Germany



The goal of the exercises: A simple Pong clone



- Writing and installing your own modules
- Parsing command line arguments
- Working with time and date
- A short introduction to pygame
- Development work flow
- MyPong – A simple Pong clone

- In the minimal case, modules consist of a single **.py** file
- Larger module are organized in directories

```
module/ __init__.py
        /file1.py
        /file2.py
        /dir1/ __init__.py
            /file3.py
```

- Importing modules

```
import module
x = module1.file1.function()
y = module1.dir1.file3.function() # Error!
```

```
import module.dir1
y = module1.dir1.file3.function1() # Works!
x = module1.file1.function2() # Works!
```

- Initialize the module
- Make module components known

```
module/__init__.py  
    /file1.py  
    /file2.py
```

```
"""module is only a stupid example of a module"""  
  
import file1  
from file2 import member1 as newname  
  
print 'module successfully initialized.'
```

- The `__init__.py` file can also be empty
- Submodule imports also execute `__init__.py` at the intermediate levels

- To make modules accessible globally, they are installed
- Installation can be carried out by the **distutils**
- All you have to do is write a **setup.py** file

```
#!/usr/bin/env python

from distutils.core import setup

setup(name          = 'MyPong',
      version       = '0.1',
      description   = 'MyPong is a clone of Pong',
      author        = 'Jochen Martin Eppler',
      url           = 'http://mindzoo.de/',
      packages      = ['mypong'],
      scripts       = ['mypong.py'])
```

- By default, **setup.py** installs globally to **/usr/lib**
- Running the installation

```
./setup.py build  
./setup.py install [--prefix=DIR]
```

- Make Python find the module in **DIR**

```
export PYTHONPATH=DIR/lib/python2.5/site-packages
```

```
import sys  
sys.path += [ 'DIR/lib/python2.5/site-packages' ]
```

- **.py** files are byte-compiled during the installation
- Help on options for the installation

```
./setup.py --help
```

- The complete command line is contained in **`sys.argv`**

```
import sys
print sys.argv
```

- Calling above program with some arguments

```
jochen@winston:~$ ./printargs.py -h arg --test
['printargs.py', '-h', 'arg', '--test']
```

- Parsing command line arguments by hand

```
for arg in sys.argv[1:]:
    if arg == '-h':
        print_usage()
    if arg == '--test':
        enable_testing()
```

- This can be a lot of work (esp. for argument combinations)

- Make parsing command line arguments easy

```
from sys import argv, exit
import getopt

shortopts = 'hv:'
longopts  = ['help', 'verbosity=']

try:
    opts, args = getopt.getopt(argv[1:], shortopts, longopts)
except getopt.GetoptError, err:
    print 'Error:', str(err)
    exit(2)

for o, a in opts:
    if o in ('-h', '--help'):
        print_usage()
        sys.exit(0)
    if o in ('-v', '--verbosity'):
        set_verbosity(a)

# Start the actual program
```

- The modules `time` and `datetime` provide functions for working with dates, times and differences thereof

```
import time, datetime

t = time.time()
print t
1212934935.78          # seconds since 1970/1/1
print time.gmtime(t)
(2008, 6, 8, 14, 22, 15, 6, 160, 0)
print time.strftime('%H:%M:%S', time.gmtime(t))
14:22:15

t0 = datetime.datetime.now()
time.sleep(6)
diff = datetime.datetime.now() - t0
print divmod(diff.seconds, 60)
(0, 6)
```

- Documentation is at <http://www.pygame.org/docs/ref/>
- Catching events

```
import pygame
from pygame.locals import *

# for queued events
for event in pygame.event.get():
    if event.type == QUIT:
        sys.exit()
    elif event.type == KEYDOWN:
        if event.key == K_SPACE:
            pause()

# concurrent key_presses aren't queued correctly
if pygame.key.get_pressed()[K_UP]:
    self.player2.up()
```

- Use the Python interpreter to test and verify your code
- Write the program simultaneously in an editor
- Perform the following in a loop:
 - Edit the code, save the file
 - Run **setup.py**
 - Check the code
- Use the integrated help system of Python

```
help(obj)           # Get help on obj
```

```
obj.__doc__        # Access the docstring of obj
```

- An interactive shell for with enhanced introspection, code highlighting and tab completion

```
jochen@winston:~$ ipython
```

```
In [1]: t = (1, 2)
```

```
In [2]: print In
```

```
['\n', u't = (1, 2)\n', u'print In\n']
```

```
In [3]: print t
```

```
(1, 2)
```

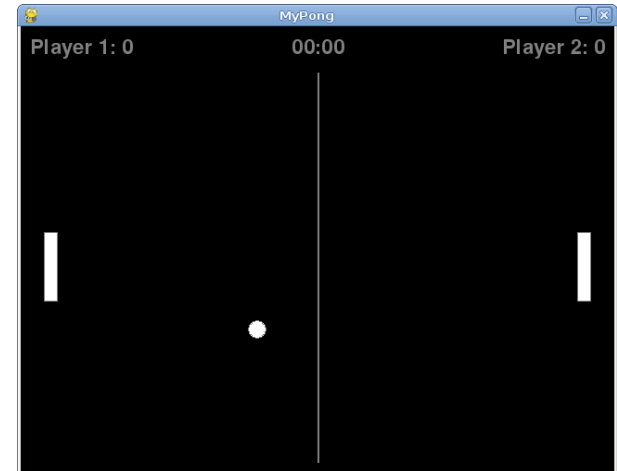
```
In [4]: t
```

```
Out[4]: (1, 2)
```

```
In [5]: print Out
```

```
{4: (1, 2)}
```

- Pong is a simple ball game, inspired by table tennis
- A module contains all the necessary files and classes for the game
- A small python program is used to orchestrate the classes according to events from the user
- You will be supplied with the canvas class, which is responsible for drawing the game onto the screen



- Create an installable module for MyPong that consists of
 - a directory skeleton for the module
 - an (empty) `__init__.py` file with a nice docstring
 - an (empty) main program file, `mypong.py` outside the module directory
- Use the `distutils` to carry out the installation
- Try installing the module and importing the module from the Python interpreter
- Hints:
 - `mkdir` creates a directory
 - `gedit` is a nice editor

- Create a main program that recognizes the following command line options
 - **--help**: Print usage information and exit
 - **--name1**: The name of the first player
 - **--name2**: The name of the second player
- The main program imports the mypong module
- It also starts the engine, see exercise 4
- Warn users that import the main program as module

```
if __name__ == '__main__':  
    # do mypong stuff  
    pass  
else:  
    print 'mypong is not meant as a module'
```

- Create a class for the ball with the following members
 - **pos**: The 2D position of the ball
 - **direction**: The movement vector of the ball
 - **move()**: Move the ball one step forward
 - **move_back()**: Move the ball one step back
 - **collide(side)**: Change the movement vector of the ball according to the side (top, bottom, left, right) of the collision
- Create a class for the players
 - **name**: The name of the player
 - **y**: The relative y-position of the bat (from [0, 100])
 - **x**: The x-position of the bat (screen coordinates)
 - **score**: The points a player achieved
 - **up()** and **down()**: Functions to move the bat
 - **reset_pos()**: A function that resets the bat position

- The game engine brings together all classes
- It contains instances of the ball, the players, and the canvas
- It runs a main loop, which updates the display and processes keyboard events in steps of ~ 50 ms
- Modify the main program to execute the engine
- User interaction: Create an event handler that allows
 - pausing the game
 - playing again after a round is finished
 - quitting the game

- The canvas provides the following members
 - **height**: the height of the window
 - **width**: the width of the window
 - **ball_radius**: The radius of the ball
 - **bat_height**: The height of the bat
 - **draw_text(string, align)**: Display a string at the top
 - **draw_message(string)**: Display a dialog window
 - **draw_bat(x, y)**: Draw a player's bat at position (x, y)
 - **draw_ball(x, y)**: Draw the ball at position (x, y)
 - **draw_center_line()**: Draw a horizontally centered line
 - **clear_screen()**: Clear the screen
 - **get_real_ypos(y)**: Return the real y position, given the relative (a number between 0 and 100)

- How about the following additions?
 - Random initial positions of the players
 - Add sound effects to the collisions
 - Store a high-score file and add a dialog to display it
 - Implement an artificial intelligence opponent
 - Add two more players at the top and bottom