

★ Back to Python

Week 4 - Pycharm, Unit Test, MVC, Project Structures

★ Today's Agenda!

- You get Pycharm, you get Pycharm, and you get Pycharm!!
- Python Unit Tests

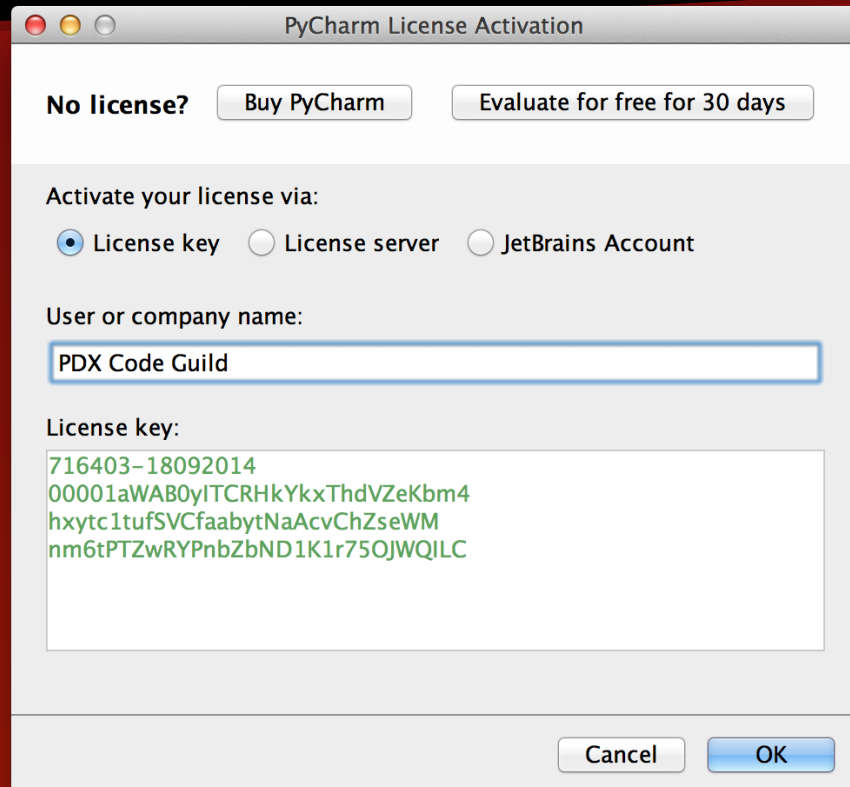
✦ Pycharm

Go to <https://www.jetbrains.com/pycharm/download/>

And download the Professional Edition of Pycharm.

★ Pycharm

Install Pycharm, and when prompted, put your license key in, like so:



The image shows a 'PyCharm License Activation' dialog box. At the top, it has three buttons: 'No license?', 'Buy PyCharm', and 'Evaluate for free for 30 days'. Below these, it says 'Activate your license via:' followed by three radio buttons: 'License key' (which is selected), 'License server', and 'JetBrains Account'. Underneath, there is a text field for 'User or company name:' containing 'PDX Code Guild'. Below that is a text area for 'License key:' containing a multi-line key: '716403-18092014', '00001aWAB0yiTCRHkYkxThdVZeKbm4', 'hxytc1tufSVCfaabytNaAcvChZseWM', and 'nm6tPTZwRYPnbZbND1K1r75OJWQILC'. At the bottom right are 'Cancel' and 'OK' buttons.

PyCharm License Activation

No license? Buy PyCharm Evaluate for free for 30 days

Activate your license via:

☒ License key ☐ License server ☐ JetBrains Account

User or company name:

PDX Code Guild

License key:

```
716403-18092014
00001aWAB0yiTCRHkYkxThdVZeKbm4
hxytc1tufSVCfaabytNaAcvChZseWM
nm6tPTZwRYPnbZbND1K1r75OJWQILC
```

Cancel OK

★ Pycharm

Add your repo to Pycharm, and allow it to use GitHub.

If you have multiple repos, add each one individually.

★ Pycharm

You should now be able to see all your files, add new ones, and also do some basic Git operations.

Notice the different color scheme for the different file types.

★ Be Defensive

Defensive Coding -

Defensively designing of programs to ensure they continue operation after unseen circumstances, such as bad input, connection issues, errors, etc.

★ Be Be Defensive

1. code for the ideal circumstance
2. code for the wrong circumstance
3. code for the **worst** circumstance

★ Keep it Tested, Keep it Safe

- Frequent testing can help ensure your code is correctly handling the various circumstances it might face.
- Manually testing can be very helpful, but automated testing makes life easier in the long run

★ Test! Test! Test!

Testing is-

the practice of writing code, that is separate from your application code, that invokes the code it tests to determine if there are any errors.

It does NOT prove if code is correct.

★ Teeeeeeeeeeeeest!

- Syntax errors:
 - unintentional misuses of the language, like the extra. in `my_list..append(foo)`
- Logical errors:
 - created when the algorithm (which can be thought of as "the way the problem is solved") is not correct

★ Automated vs Manual

Manual Testing - A human drives the test, running the program and testing various circumstances

Automated Testing - A program runs a series of predefined tests that check for correct functionality under various circumstances

★ Sidenote - Quality Assurance

QA, as it's often referred to, is one of the most valuable teams a company can have.

They design and implement tests to ensure that the product stands up to the standards outlined at design.

★ Good Software Developers Test

- The ability to write good tests is a hiring point for a lot of companies
- A developer that knows how to write unit tests for their code is much more valuable than one who does not

★ Unit Tests

- a test which tests a single *unit* of code, in isolation of the rest of the code
- testing in isolation ensures that the results you get are due to the code you're testing, not another part of your application

★ unittest

- Python has a built in test module called unittest
- it automagically knows where your tests are, so long as you name them accurately

★ unittest - Best Practices

- Each unittest should test just one unit of your code, such as a function
- It should test the ideal circumstances, less than ideal circumstances, and then non-ideal circumstances
- It should test just the LOGIC of that unit, not HOW it works

✦ unittest - Test Case

A Test Case should contain a series of tests about the one thing you're testing.

If you're testing a single function, all the tests in your test case would test that function.

★ unittest - Tests

Your Test Case will contain several tests that test the various circumstances that your unit should handle.

Your tests are what tests - your Test Case just contains them and defines what unit you're testing.

★ unittest - Naming Best Practices

Different companies will have different naming rules; My rules are:

- should start with “test_”
- should tell you the unit it’s testing
- all lowercase

example: `test_angry_dice_roll_dice`

★ unittest - Test Case Names

Your Test Cases should have names that follow the class naming rules for PEP8, and identify the class as a test class.

AngryDiceRollDiceTest()

DieRollTest()

★ Testing Print Statements

- Leverage Python's Mock module to mock being stdout
 - stdout is "standard out"
- Leverage Python's Mock module to mock being a response to input() as well
 - builtins.input

✦ Mock sys.stdout

```
import unittest
from angry_dice import AngryDiceGame
from unittest.mock import patch
from io import StringIO

# Mocks being sys.stdout and will store what is print()'d
# into mock_stdout
@patch('sys.stdout', new_callable=StringIO)
def test_both_die_angry(self, mock_stdout):
    """Set both dice to ANGRY and check that the game resets their game."""
    self.angry_game.die_a.setDieFaceValue("ANGRY")
    self.angry_game.die_b.setDieFaceValue("ANGRY")

    angry_text = "WOW, you're ANGRY!\nTime to go back to Stage 1!\n"

    self.angry_game.check_angry()

    self.assertEqual(mock_stdout.getvalue(), angry_text)
    self.assertEqual(self.angry_game.current_stage, 1)~
```

✦ Mock builtins.input

```
import unittest
from angry_dice import AngryDiceGame
from unittest.mock import patch

# Will mock the input for 1 input prompt
@patch('builtins.input', return_value='ab')
def test_input_a_and_b(self, inputted_value):
    dice_to_roll = self.angry_game.determine_roll()

# Will mock the input for len(passed_list) input prompts
@patch('builtins.input', side_effect=['ab', 'b', 'a'])
def test_input_a_and_b(self, inputted_value):
    dice_to_roll = self.angry_game.determine_roll()
```