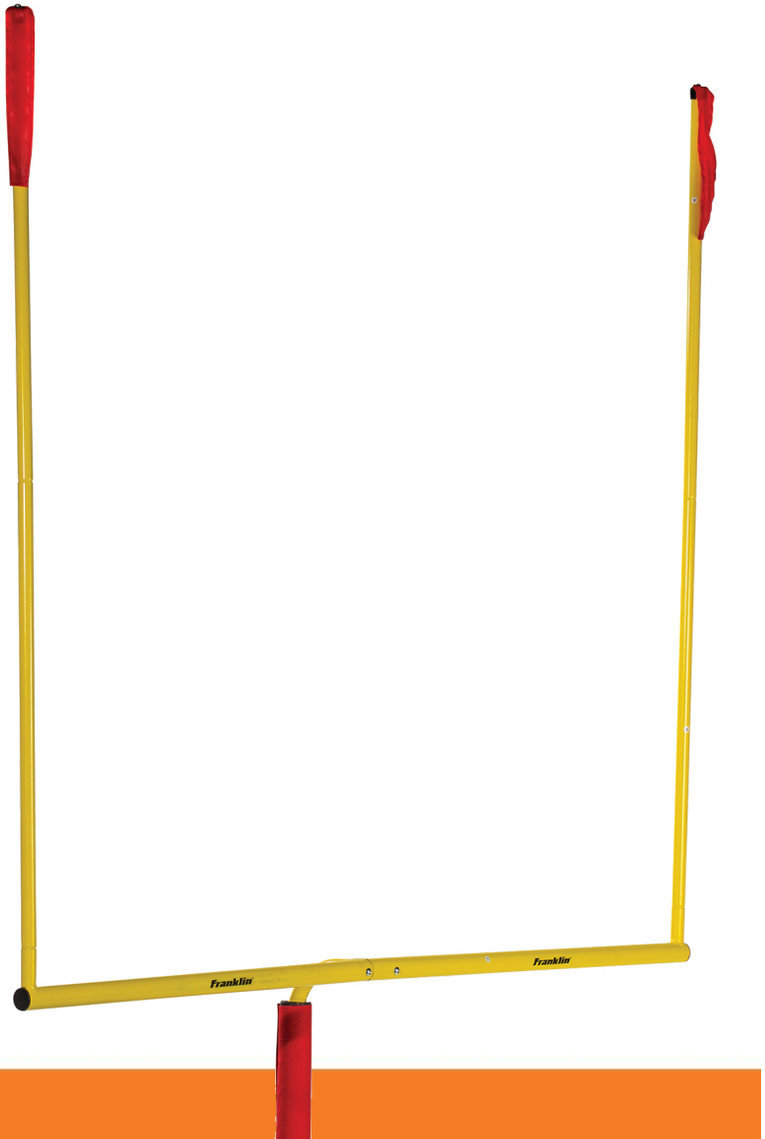
Quality

Develop Intelligence

Goals

1. Name 2 things banned in strict mode
2. Explain how to mock a cmdlet using Pester
3. Name 2 unit testing best-practices



Roadmap

1. Sloppy & Strict
2. Lint Extravaganza
3. Using Pester
4. Best Practices

Develop
Intelligence

Sloppy & Strict

Develop
Intelligence

Preface

- Pure Powershell lets you screw up silently
- Best practices
  - Add your own guardrails
  - Fail early

Motivation

Develop Intelligence

This doesn't work, but there's no error

```
function Get-Message{
  param(
    [string]$Path
  )
  return "Here is your path: $File"
}

Get-Message c:/temp
```

(In sloppy mode, that is.)

About Strict Mode

- Strict mode changes semantics to avoid common potential errors
  - Throws more exceptions instead of failing silently
  - Disallows certain constructs
- Forbids
  - Using uninitialized variables
  - Referencing to non-existent properties
  - Referencing invalid index in an array
  - Improper function syntax

Example

Now my script fails early

```
1  Set-StrictMode -Version Latest
2
3  function Get-Message{
4    param(
5      [string]$Path
6    )
7    return "Here is your path: $File"
8  }
9
10 Get-Message c:/temp
```

Advice

- Always be strict
- **Don't** use `-Version Latest` in production

Develop
Intelligence

Lint Extravaganza

Develop Intelligence

Overview

- Linting is static code analysis
- Advantages:
  - Catch possible errors
  - Enforce consistent style
  - Automatically fix problems
- Disadvantages:
  - Irritating and overbearing out of the box
  - Tools are tempermental

ScriptAnalyzer

- Current standard for Powershell
- Plays well with VSCode

Usage

Command

```
Invoke-ScriptAnalyzer -Path ./labs -Recurse -Settings ./PSScriptAnalyze
```

Output

```
RuleName                                Severity     ScriptName Line  Mess
--------                                --------     ---------- ----  ----
PSUseProcessBlockForPipelineCommand     Warning      Get-Square 6     Comm
                                                                      proc
PSUseDeclaredVarsMoreThanAssignment     Warning      Main.ps1   1     The
PSUseDeclaredVarsMoreThanAssignment     Warning      Get-Fortun 1     The
PSUseConsistentIndentation              Warning      Get-Users. 16    Inde
PSUseConsistentIndentation              Warning      Get-Users. 17    Inde
PSUseConsistentWhitespace               Warning      Get-Users. 17    Use


PSUseConsistentWhitespace               Warning      Get-Users. 17    Use
```

Auto-Fixing Problems

Develop Intelligence

- Tries to fix:

```
Invoke-ScriptAnalyzer -Path ./labs `
  -Recurse `
  -Settings ./PSScriptAnalyzerSettings.psd1 `
  -Fix
```

Linting Best Practices

Develop Intelligence

1. Lint automatically-- in the IDE and CI builds
2. Lint errors fail the build
3. Automatically and transparently fix minor style issues, e.g. tabs v spaces
4. Use lint warnings judiciously--

- They add visual clutter, distracting from actual problems
- They’re often ignored anyway

Develop
Intelligence

Using Pester

Develop Intelligence

Definitions

***Unit test**: low-level test case written in the same language as the production code, which directly access its objects and members.*

- Unit testing is surprisingly controversial
- Probably because it’s hard to do well
- Think of testing as a skill-- like SQL

Common Complaints

- Time to write
- Time to maintain
- Tests are worthless if they don’t get run

Reasons to Test

1. Catch bugs - Especially important without static typing
2. Refactor with confidence
3. Living documentation
4. ***Encourage good design***

Especially with Dynamic Typing

- Typed languages are better
  - TypeScript eliminates 38% of bugs
- Tests are especially important in Powershell
  - No compile-time help
  - No expressivess through the type

About Pester

- Industry standard
- Easy
- Includes a mock framework too!

Big Four

Develop Intelligence

Four most important functions:

- **`Describe`** encapsulates a test suite
- **`It`** creates a ‘spec’, does the actual work of testing
- **`Should`** asserts expected behavior
- **`Mock`** fakes out dependencies

Example


```
BeforeAll {
   Import-Module -Force $PSCommandPath.Replace('.Tests.ps1', '.ps1')
}
Describe 'Get-Power' {

   It 'Should be 4 for 2**2' {
      Get-Power 2 2 | Should -Be 4
   }

   It 'Should be 0 for 0**0' {
      Get-Power 0 0 | Should -Be 0
   }

}
```

Idiom: Arrange, Act, Assert

Organize a spec--aka test function-- into three sections:

1. **Arrange** all necessary preconditions, inputs
2. **Act** on the functionality under test
3. **Assert** that the results are as expected

Dealing With Exceptions

Develop Intelligence

```
Describe 'Get-Quotient' {

   It 'Should be 1 for 2/2' {
      Get-Quotient 2 2 | Should -Be 1
   }

   It 'Should throw on 1/0' {
      { Get-Quotient 1 0 } | Should -Throw
   }

}
```

Dealing with Dependencies

- Creating a double is often necessary to avoid IO
  - Double object stands in for a dependency
- Doubles come in flavors:
  - A mock verifies behaviour based on expectations
  - A stub just sits there

Mocking Dependencies

Cmdlet

```
function Restart-Themes {
   Stop-Service 'Themes'
   Start-Service 'Themes'
}
```

Test

```
Describe 'Restart-Themes' {
   It 'Should call Stop-Service and Start-Service' {
      Mock Stop-Service {} -Verifiable
      Mock Start-Service {} -Verifiable

      Restart-Themes
      Should -InvokeVerifiable
   }
```

Setting up State

- These things:
  - beforeEach
  - beforeAll
  - afterEach
  - afterAll
- Suggest tight-coupling
- **Best practice:** Use a pure function wherever possible. They’re easier to test, since there’s no state to set up, and harder to screw up.

Develop
Intelligence

Best Practices

Develop Intelligence

Unbreakable Spec Rules

1. **No I/O**
   - No database
   - No disc
   - No network
2. **Test 1 thing**
   - High cohesion
   - Ideally a pure function
3. **Run fast**
   - Well under 1s

Don’t Test Stupid Stuff

- Getters and Setters for instance
- The framework you’re working with
- Browser/language behavior

Test from Day 0

- Good unit testing is hard to add to existing, non-tested code
- Estimates include time to build tests
- Consider a document called “Done is...”

TDD & YPP

Test Driven Development encourages you to write tests first

1. Stub-out the feature you're building (Tests pass)
2. Write a test for the stubbed-out feature (New test fails)
3. Implement your new feature. You're done when the test pass.

Do what works for you

My favorite thing: Bug and tests

- When you get a bug, create a test that exposes the bad behavior
  - i.e. before you try to fix anything you should have a failing test
- Idea is: bugs in the wild mean your unit tests failed.

Build Tests into Processes

Develop Intelligence

Tests should:

- **Always pass** A failing test should be the same as a broken build
- **Run automatically** Build it into your process

Metric: Coverage

Develop Intelligence

The 2 unit test metrics that matter:

- **Coverage** - Usually it’s blocks of code, rather than lines
- **Coverage over time** - Should be going up

Coverage goals are arbitrary:

- Even 100% coverage never exercises all permutations
- But set a coverage goal - Maybe 40-60%
- Think of it like your 5 servings of fruits and vegetables

Metric: Execution time

- Run tests asynchronously where possible
- Set an upper limit on per-test execution time
  - Encourages tightly-focused unit tests
  - Ensures the entire suite runs fast, which means the entire suite gets run.

Lab: Fortune

- Read instructions: $/labs/file-fortune/README.md
- Build the applications starting with 'startingpoint'

Develop
Intelligence

Review

1. Name 2 things banned in strict mode
2. Explain how to mock a cmdlet using Pester
3. Name 2 unit testing best-practices