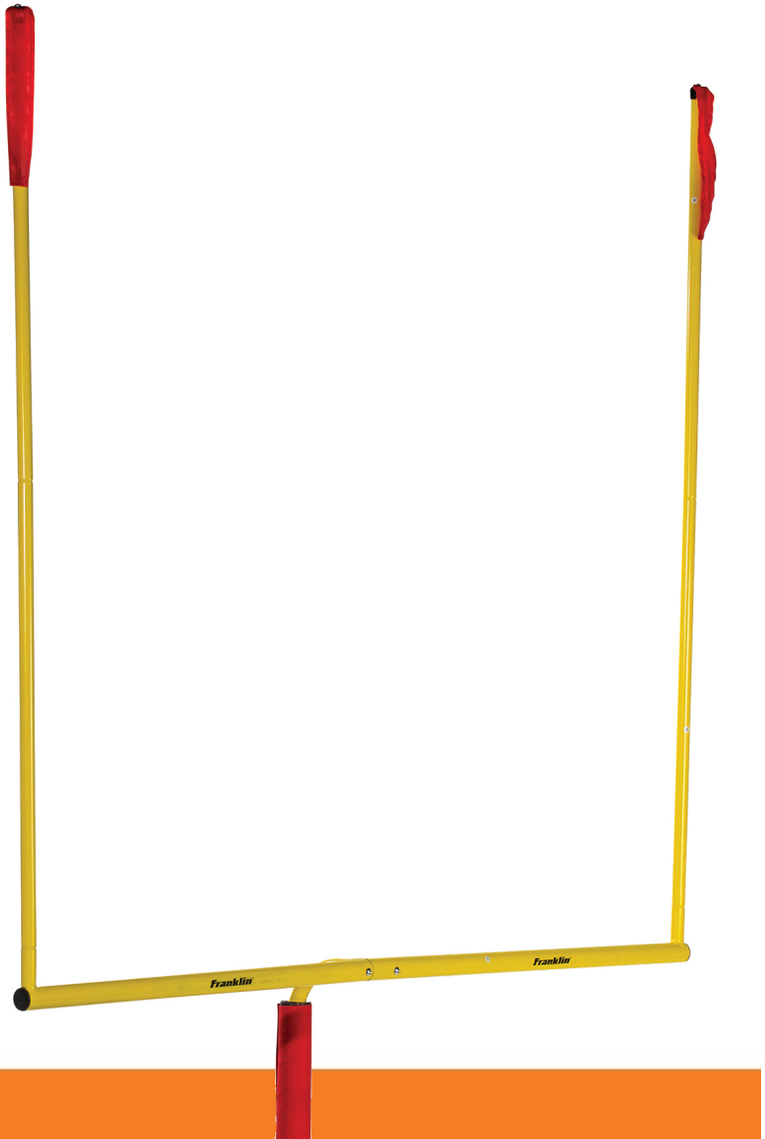
Collections

Develop Intelligence

Goals

1. Describe how to slice an array
2. Explain Enumeration
3. Give the PROs and CONs of dictionaries



Roadmap

1. Arrays
2. Enumeration
3. Hash Tables

Develop
Intelligence

Arrays

Develop Intelligence

Overview

- Based on .NET `Object[]`
  - Contiguous memory
  - Heterogeneous
- Powershell makes them
  - Ergonomic
  - Sometimes weird

Syntax

Develop Intelligence

```
$as = 1,2,3
$bs = @(1, 2, 3)
$cs = 1..3

$xs = 'danish','strudle',3.14
$ys = @(1)
$zs = ,'chicken'
```

Nested Syntax

- aka 'Jagged' is a kind of multi-dimensional array
- Width and depth may not be the same

```
$lists = @(
  @('a','b','c'),
  @('A','B','C'),
  @('X','Y','Z')
)

foreach($list in $lists){
  Write-Host -ForegroundColor Green "Here's a list:"
  Write-Host $list
}
```

Indexing

Develop Intelligence

- Zero-based

```
$colors = 'red','green','purple','orange'
$first =  $colors[0]
$last = $colors[$colors.Length-1]
$alsoLast = $colors[-1]
```

Mutation

Develop Intelligence

```
$colors = @()

# Actual mutation
$colors[0] = 'forestGreen'
$colors.Clear()

# Fake mutation
$colors += 'green'
$colors += 'blue', 'brown'
```

Fancy Indexing

- The subscript of an array... can be another array

```
#Arbitrary elements
$letters = 'a', 'b', 'c', 'd', 'e'
$letters[0, 2, 4]

#Array as an index
$places = 'ocean', 'mountains', 'desert', 'jungle', 'forest', 'city'
$indexes = 0, 3, 0, 4
$places[$indexes]
```

Slicing

- Also zero based
- Reversable
- Inclusive index
- Operator + enables multi-array subscripts

```
$colors = 'red','green','purple','orange', 'blue'
$head =  $colors[0]
$tail = $colors[1..-1]

$pairs = $xs[0..1+-2..-1]
```

Destructuring

- Use assignment to grab elements from the array

```
$xs = 1, 2, 3
$head, $tail = $xs
"Head: $head"
"Tail: $tail"
```

```
$brown = 165, 42, 42
$red, $green, $blue = $brown
```

Gotcha: Resizing

Develop Intelligence

- The += operator creates a whole new array
- This gets expensive for long lists
- `ArrayList` can be quicker
- **But** it's less ergonomic

```
$colors = [System.Collections.ArrayList]::new()
$colors.Add('red')
$colors.Add('green')
$colors.Add('blue')
```

Develop
Intelligence

Enumeration

Develop Intelligence

Overview

When executing a pipeline, PowerShell automatically enumerates any type that implements the IEnumerable interface and sends the members through the pipeline one at a time. ***The exception is [hashtable]****...*

- *Strings are also not enumerated

Example

```powershell
$thingArray = 'dog', 'planet', 'rag'

$thingTable = @{
  items='dog','planet','rag'
}

Write-Host "`nThings:" -ForegroundColor Green
$thingArray | Write-Output

Write-Host "`nMore Things:" -ForegroundColor Green
$thingTable | Write-Output
```

Opting Out

*However, there's an important difference. When you **pipe** multiple objects to a command, PowerShell sends the objects to the command one at a time. When you use a **command parameter**, the objects are sent as a single array object. **This minor difference has significant consequences.***

Example

- If `process` looks weird, pretend it's magic-- for now.

```
function Get-Count($Target){
  process{
    if($Target -is [array]){
      return "Got an array with $($Target.Length) items."
    }
    return "Got 1 item."
  }
}
Write-Host "`nPiped things:" -ForegroundColor Green
1,2,3 | Get-Count

Write-Host "`n Param things:" -ForegroundColor Green
Get-Count 1,2,3
```

Gotcha: Automatic Enumeration

```powershell
#1
'a','b','c' | Get-Member

#2
function Get-Self([array]$Self){
  return $Self
}
(Get-Self @('x','y','z')).GetType()
(Get-Self @('x')).GetType()
```

Example #2

```powershell
function Get-Self{
  param(
    $Self
  )
  return $Self
}

$a = 1, 2, 3
$b = Get-Self $a

$a[0] = 'chicken'

"`nHere is a: $a"
"`nHere is b: $b"
```

Return and Pipeline

When you return a collection from your script block or function, PowerShell automatically unrolls the members and passes them one at a time through the pipeline. This is due to PowerShell's one-at-a-time processing.

Fix: NoEnumerate

```powershell
function Get-Self{
  param(
    $Self
  )
  Write-Output $Self -NoEnumerate
}

$a = 1, 2, 3
$b = Get-Self $a

$a[0] = 'chicken'

"`nHere is a: $a"
"`nHere is b: $b"
```

Takeaway

- **Piped arguments** get passed one at a time
  - Opt-out by using parameters instead
- **Returned values** get passed one at a time
  - Opt-out by using `Write-Output -NoEnumerate`

Develop
Intelligence

Hash Tables

Develop Intelligence

Overview

- Associates keys with values
- Also known as
  - Map
  - Dictionary
  - Associative array
- Rules
  1. Keys should be immutable-- usually strings
  2. Values are anything

Syntax

Develop Intelligence

```powershell
# Multiline
$words = @{
  fancy = 'plethora', 'eschew'
  plain = 'big', 'lots', 'some'
}

# Oneline
$user = @{id=12;name='jbloggs'}

# Setting things later
$user.createdOn = Get-Date

# Alternative syntax via subscript
$user['createdOn'] = Get-Date
```

Non-String Keys

- Literal syntax doesn't work

Bork!

```
$enrollmentByGrade=@{
  4=233
  5=232
}
```

Ok

```
$gradeEnrollment=@{}
$gradeEnrollment.Add(4, 233)
$gradeEnrollment.Add(5, 232)
```

Complaints

Develop Intelligence

- No metadata!

```
$users = @(
  @{id=11;login='jbloggs';roles='admin','remoteUser'},
  @{id='yes';login='jdoe';roles='admin','contractor'}, #Type issue
  @{id=13;login='ksun';} #Missing field
)
```

Alternatives (Preview)

- Make a simple class
- Cast from hash table
- Stay tuned for more!

```powershell
class Account{
  [int]$id
  [string]$login
  [string[]]$roles
}

$users = @(
  [Account]@{id=11;login='jbloggs';roles='admin','remoteUser'},
  [Account]@{id=12;login='jdoe';roles='admin','contractor'},

  [Account]@{id=13;login='ksun';}
```

Develop
Intelligence

Review

1. Describe how to slice an array
2. Explain Enumeration
3. Give the PROs and CONs of dictionaries