Что такое Windows PowerShell и с чем его едят? Часть 3: передача параметров в скрипты и функции, создание командлетов

* [Блог компании RUVDS.com](https://habr.com/ru/company/ruvds/),
* [Системное администрирование](https://habr.com/ru/hub/sys_admin/),
* [Системное программирование](https://habr.com/ru/hub/system_programming/),
* [PowerShell](https://habr.com/ru/hub/powershell/),
* [Разработка под Windows](https://habr.com/ru/hub/win_dev/)

[](https://habr.com/ru/company/ruvds/blog/493366/)  
  
Во [**второй части цикла**](https://habr.com/ru/company/ruvds/blog/490924/) рассматривались основы языка программирования PowerShell, а сейчас стоит разобраться с использованием написанного на нем кода для задач администрирования. Самый очевидный способ это сделать — запустить сценарий. Кроме него существует возможность создания собственных командлетов.

Оглавление:

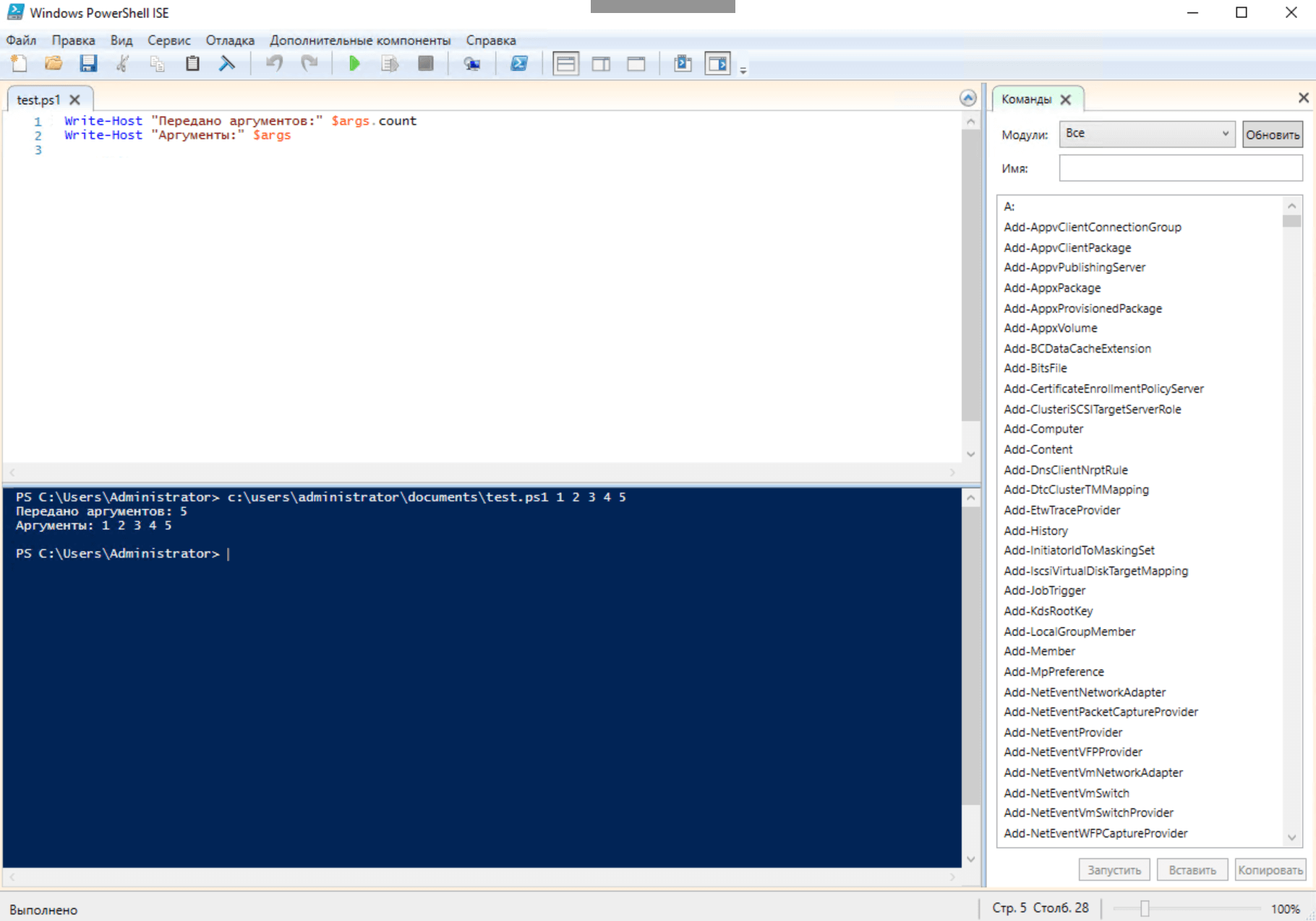
[Позиционные параметры](https://habr.com/ru/company/ruvds/blog/493366/#section1)  
[Блок Param()](https://habr.com/ru/company/ruvds/blog/493366/#section2)  
[Дополнительные атрибуты параметров](https://habr.com/ru/company/ruvds/blog/493366/#section3)  
[Передача параметров через конвейер](https://habr.com/ru/company/ruvds/blog/493366/#section4)  
[Структура тела функции](https://habr.com/ru/company/ruvds/blog/493366/#section5)  
[Атрибут [CmdletBinding()] и расширенные функции](https://habr.com/ru/company/ruvds/blog/493366/#section6)  
[Модули сценариев и создание командлетов](https://habr.com/ru/company/ruvds/blog/493366/#section7)

Позиционные параметры

В сценарии и [функции](https://habr.com/ru/company/ruvds/blog/490924/#section13) можно передавать позиционные параметры (аргументы), значения которых записываются во встроенную переменную $args. Этот одномерный [массив](https://habr.com/ru/company/ruvds/blog/490924/#section11) не требует предварительного объявления, а область его видимости ограничивается скриптом или функцией. Для примера запустим простейший сценарий:

Write-Host "Передано аргументов:" $args.count

Write-Host "Аргументы:" $args

  
  
В функциях позиционные параметры используются аналогично:

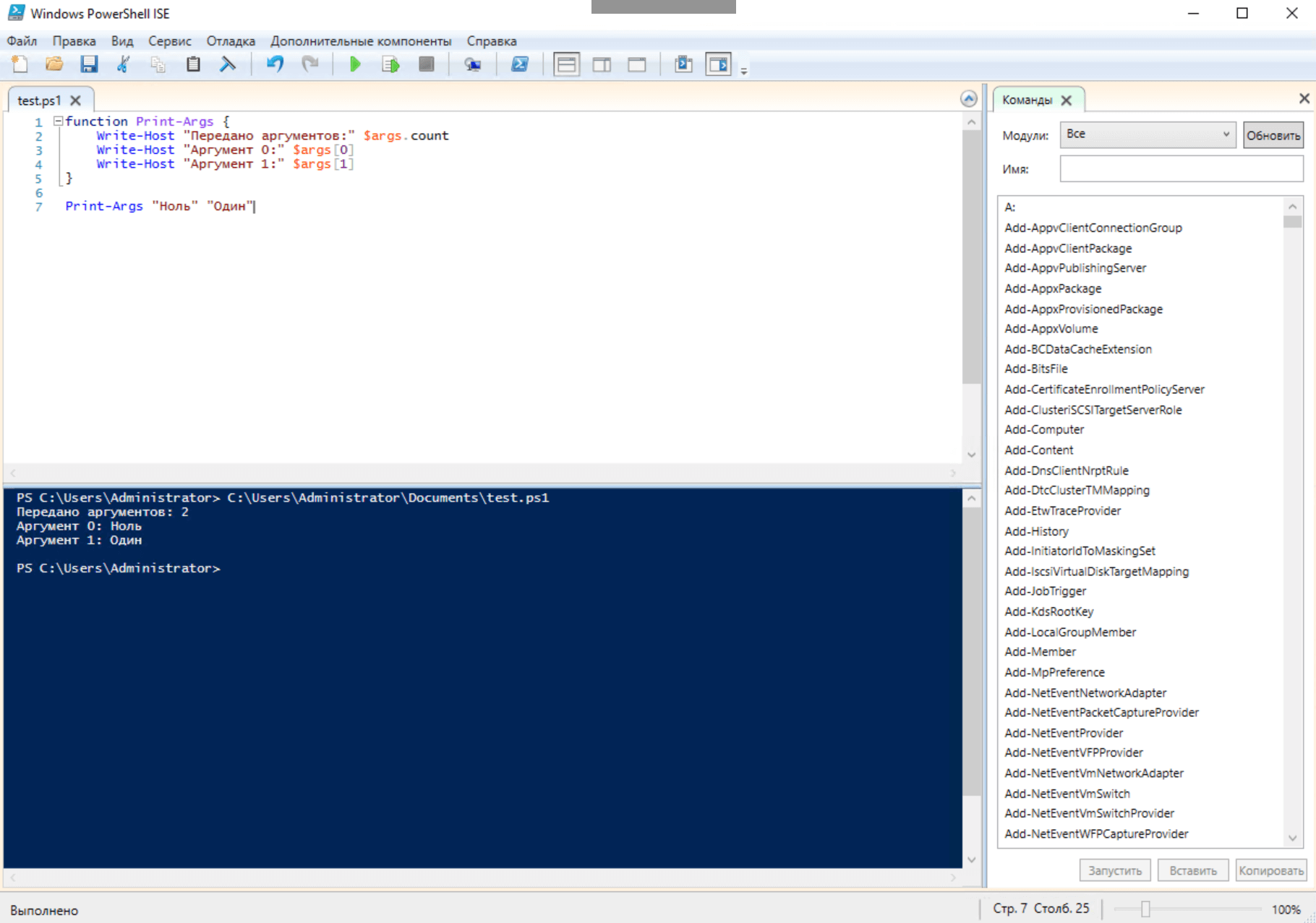
function Print-Args {

Write-Host "Передано аргументов:" $args.count

Write-Host "Аргумент 0:" $args[0]

Write-Host "Аргумент 1:" $args[1]

}

Print-Args «Ноль» «Один»  
  
Обратите внимание, что при вызове Print-Args мы не ставим запятую между параметрами: в функцию передается не массив, а отдельные значения, которые записываются в одномерный массив $args — его область видимости ограничена телом функции.  
  
  
  
Описанный выше способ позволяет передать в сценарий или функцию любое количество параметров, но при вызове необходимо соблюдать порядок их следования, а обращаться к ним можно только по индексу массива — это не всегда удобно.

Блок Param()

В сценариях и функциях намного удобнее использовать именованные параметры. В [предыдущей статье](https://habr.com/ru/company/ruvds/blog/490924/#section13) мы рассказывали об одном способе их описания:

function test ($arg0, ..., $argN)

{

}

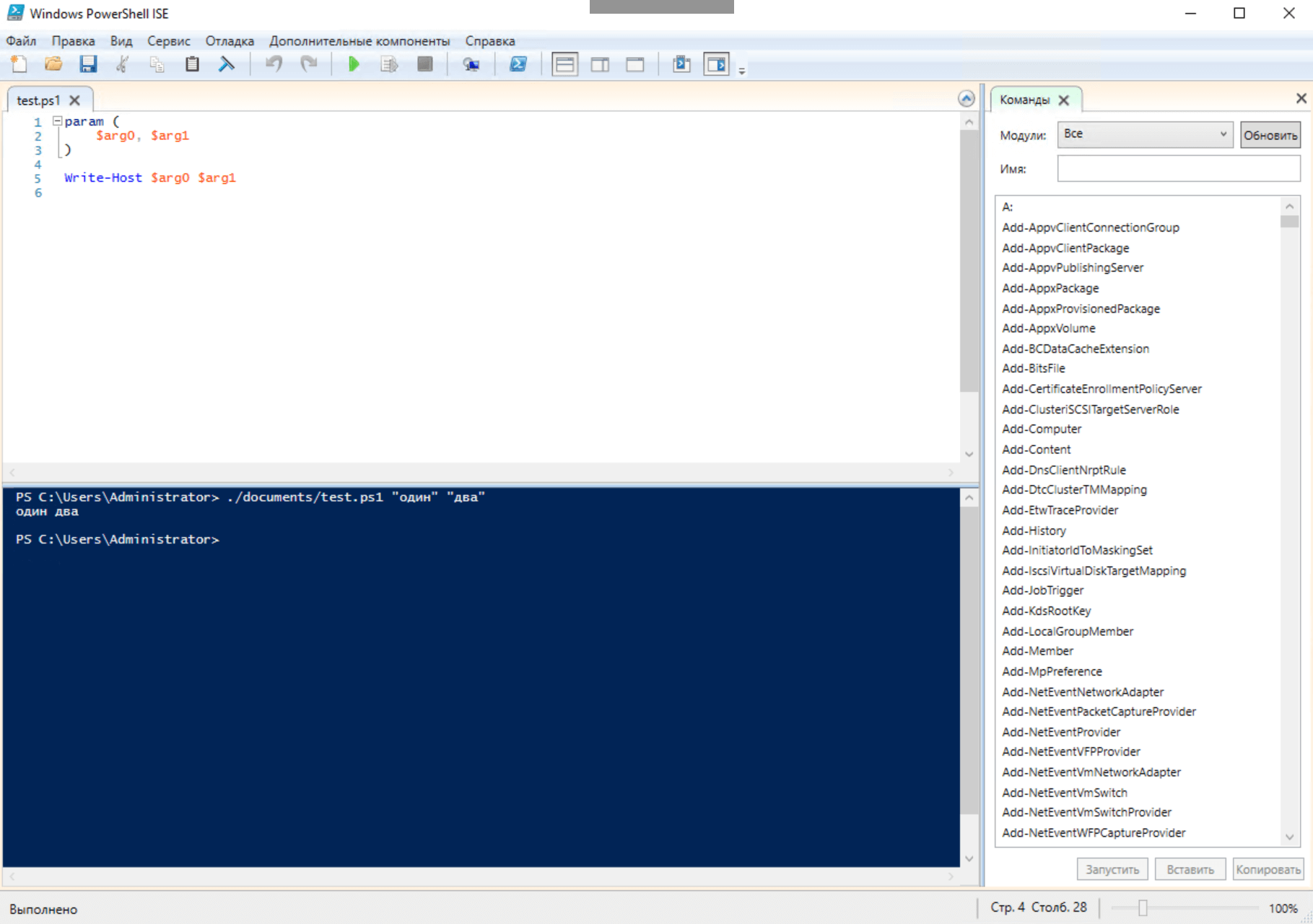
Подобный синтаксис привычен разработчикам, но при вызове функции параметры (если они есть) разделяются пробелами, а не заключаются в круглые скобки — возникает некоторый диссонанс. Такова специфика shell-языков: для работы с командным интерпретатором в интерактивном режиме пробелы между значениями намного удобнее. Вызов **test($value0)** также корректен, но параметром при этом является все выражение в скобках, т.е. **($value0)** вместо **$value0**. Передать таким способом несколько параметров уже не выйдет. В результате вызова **test($value0, $value1)** функция получит только один — массив из двух элементов со значениями **$value0** и **$value1**.  
  
Корпорация Microsoft рекомендует использовать блок Param() — этот синтаксис более универсален и позволяет задавать не только аргументы функций, но и параметры сценариев:

param (

$arg0, $arg1

)

Write-Host $arg0 $arg1

  
  
В теле функции это выглядит так:

function test {

param (

$arg0, $arg1

)

}

Если список аргументов функции невелик, блок Param() только загромоздит конструкцию, но во многих случаях он делает код более читаемым и является помимо прочего элементом хорошего стиля программирования.

Дополнительные атрибуты параметров

При описании аргументов функции или параметров скрипта можно задать их дополнительные атрибуты. Самый простой пример — принудительная установка типа:

param([int]$arg0)

или

function test ([int]$arg0) {

}

Помимо приведения типов можно использовать атрибут [parameter()]:

param(

[parameter(Argument1=value1, Argument2=value2)]

$ParameterName

)

С его помощью нетрудно сделать параметр обязательным. Обратите внимание на одновременное использование нескольких атрибутов — в этом случае они идут друг за другом:

param([parameter(Mandatory=$true)][int]$arg0)

или

function test ([parameter(Mandatory=$true)][int]$arg0) {

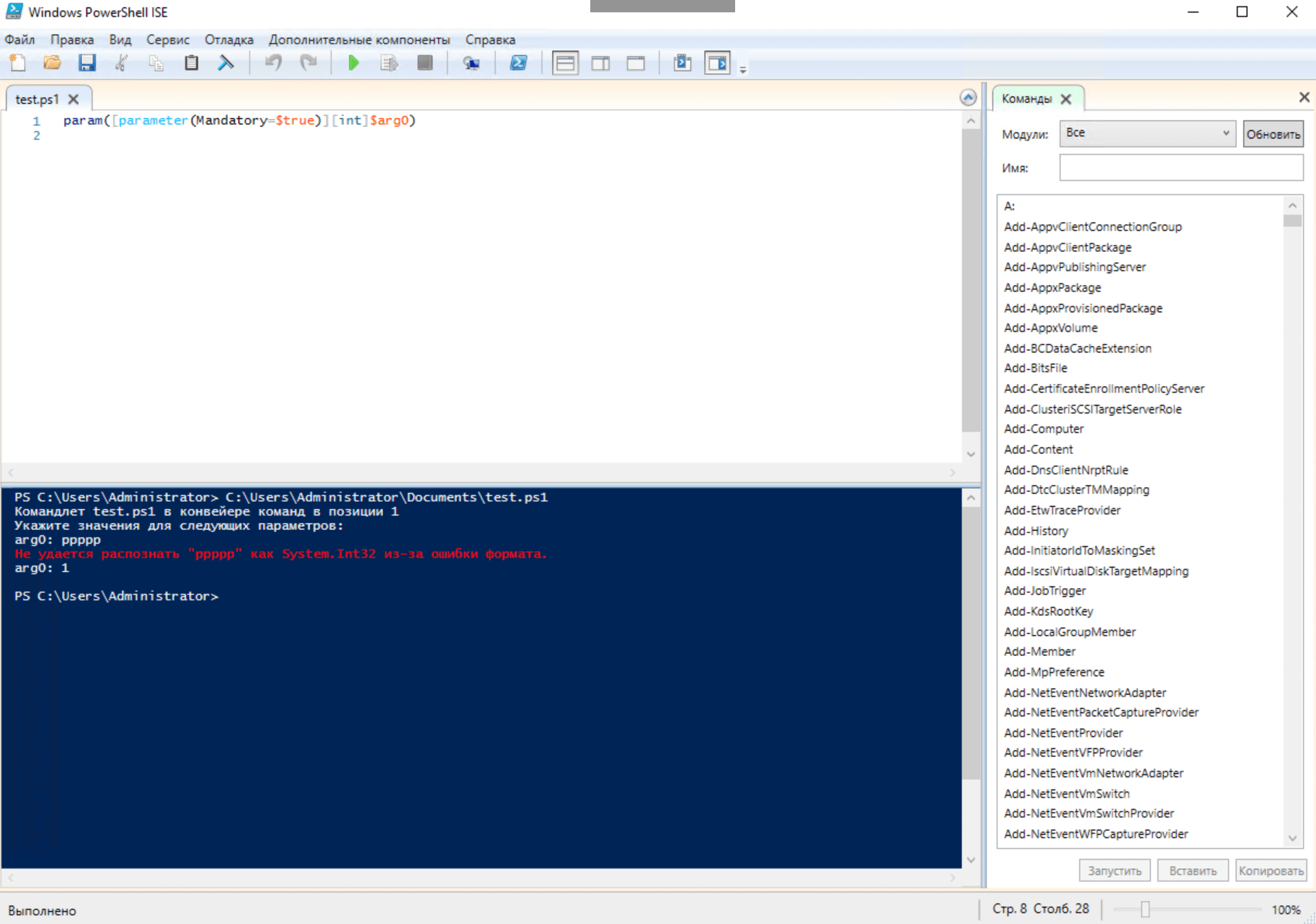
}

или

function test {

parameter([parameter(Mandatory=$true)][int]$arg0)

}

  
  
Position позволяет указать порядок следования параметра (по умолчанию он соответствует порядку объявления):

param(

[parameter(Mandatory=$true, Position=0)]

[int]

$arg0,

[parameter(Position=1)]

[string]

$arg1,

[parameter(Position=2)]

[array]

$arg2

)

У атрибута [Parameter()] есть и другие аргументы, полный список которых доступен [на сайте](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_functions_advanced_parameters) Microsoft. Там же описаны прочие атрибуты, с помощью которых можно провести валидацию переданных значений, проверить их с использованием регулярных выражений и т.д. Перечислим некоторые:  
  
**[Alias()]** устанавливает псевдоним для параметра:

param(

[parameter(Mandatory=$true)]

[alias("ARG","ArgumentName")]

[string[]]

$arg0

)

Оператор приведения типов [string[]] означает, что значение параметра — строковый массив.  
  
**[AllowNull()]** разрешает $null в качестве обязательного параметра:

param(

[parameter(Mandatory=$true)]

[AllowNull()]

[string]

$arg0

)

**[AllowEmptyString()]** разрешает пустую строку в качестве обязательного параметра:

param(

[parameter(Mandatory=$true)]

[AllowEmptyString()]

[string]

$arg0

)

**[AllowEmptyCollection()]** разрешает пустой массив в качестве обязательного параметра:

param(

[parameter(Mandatory=$true)]

[AllowEmptyCollection()]

[string[]]

$arg0

)

**[ValidatePattern()]** проверка с использованием регулярного выражения:

param(

[parameter(Mandatory=$true)]

[ValidatePattern("[0-9][0-9][0-9][0-9]")]

[string[]]

$arg0

)

**[ValidateLength()]** проверяет длину строкового параметра:

param(

[parameter(Mandatory=$true)]

[ValidateLength(1,10)]

[string]

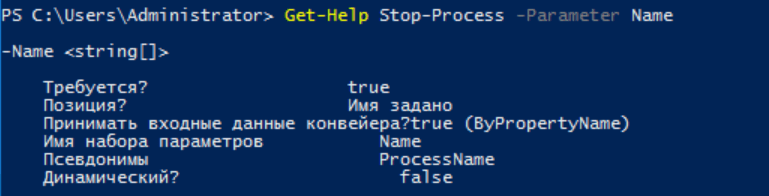
$arg0

)

Передача параметров через конвейер

В [первой статье](https://habr.com/ru/company/ruvds/blog/487876/) цикла мы рассказывали о возможности передачи данных в командлеты через конвейер (pipeline). В PowerShell командлеты и функции возвращают объекты или массивы объектов (результаты стейтментов), а также получают их на входе. Чтобы это увидеть, препарируем один из командлетов при помощи Get-Help:

Get-Help Stop-Process -Parameter Name

  
  
Через конвейер можно принять значения параметров, для которых установлены соответствующие атрибуты (ByValue и/или ByPropertyName). В первом случае параметру будет сопоставлен поступивший по конвейеру объект при условии, что его тип соответствует ожидаемому. Во втором значением параметра будет свойство входящего объекта, имя которого соответствует имени или псевдониму этого параметра. Для установки атрибутов используется [parameter()] с булевыми аргументами ValueFromPipeline и ValueFromPipelineByPropertyName, значение которых по умолчанию равно $false:

param(

[parameter(Mandatory=$true,

ValueFromPipeline=$true)]

[string[]]

$Name

)

или

param(

[parameter(Mandatory=$true,

ValueFromPipelineByPropertyName=$true)]

[string[]]

$Name

)

ValueFromPipelineByPropertyName обычно используют при необходимости передать несколько параметров, чтобы не возникало путаницы, при этом аргумент можно применять одновременно с ValueFromPipeline:

param(

[parameter(Mandatory=$true,

ValueFromPipeline=$true,

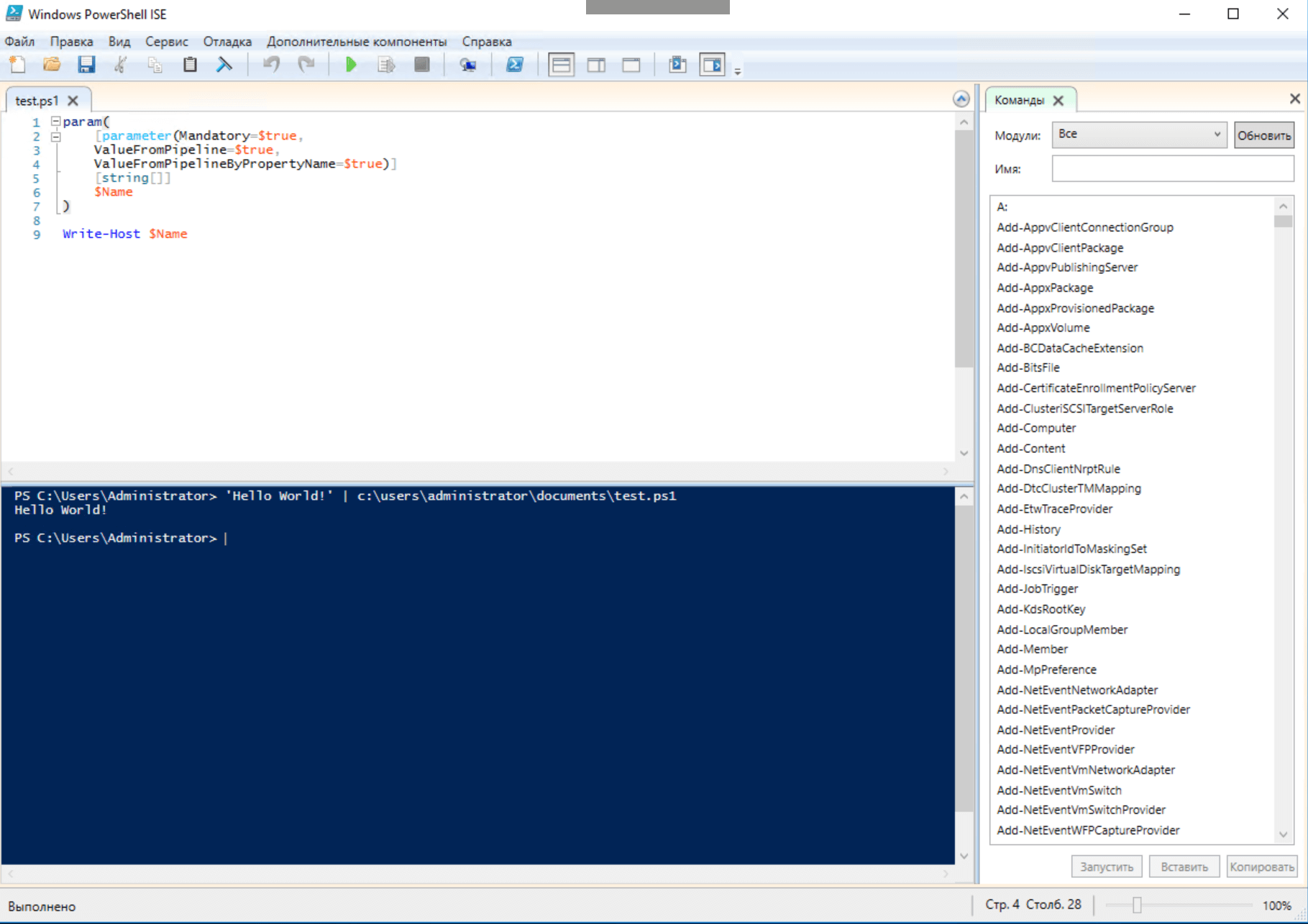
ValueFromPipelineByPropertyName=$true)]

[string[]]

$Name

)

Write-Host $Name

  
  
Как видите, получать параметры через конвейер могут и скрипты, но все же практическое применение описанных выше атрибутов характерно скорее для [расширенных функций](https://docs.google.com/document/d/1S4yewdXQ4pisph2J7GT9WafySzPIzZG9_McNHgB83es/edit?pli=1#heading=h.o9qzjk2d38gv), которые будут рассмотрены ниже.

Структура тела функции

В языке PowerShell функция может включать три необязательных блока заключенного в операторные скобки кода — Begin, Process и End. Выглядит она примерно так:

function test

{

param()

begin {}

process {}

end {}

}

Первым однократно выполняется блок Begin, причем если параметры передаются в функцию через конвейер, код запустится еще до поступления первого объекта на обработку. Переменные $\_ и $PSItem в блоке Begin в этом случае не будут содержать значений. Если же функция вызывается с использованием явным образом заданных параметров, они будут доступны и в блоке Begin, поскольку нет необходимости ожидать получения объектов из конвейера. Далее выполняется блок Process: если параметры передаются через конвейер, он будет поочередно запущен для каждого объекта. В случае явным образом заданных параметров блок Process запустится только один раз. Завершается работа функции однократным выполнением блока End. Очевидно, что использование этих конструкций оправдано, только если функция может принимать объекты из конвейера:

function test

{

param(

[Parameter(ValueFromPipeline)]

[string[]]

$Param1,

[string]$Param2

)

begin

{

Write-Host "Блок Begin"

Write-Host " Первый параметр (через pipeline):" $Param1

Write-Host " Второй параметр (аргумент функции):" $Param2

}

process {

Write-Host "Блок Process"

Write-Host " Первый параметр (через pipeline):" $Param1

Write-Host " Второй параметр (аргумент функции):" $Param2

}

end

{

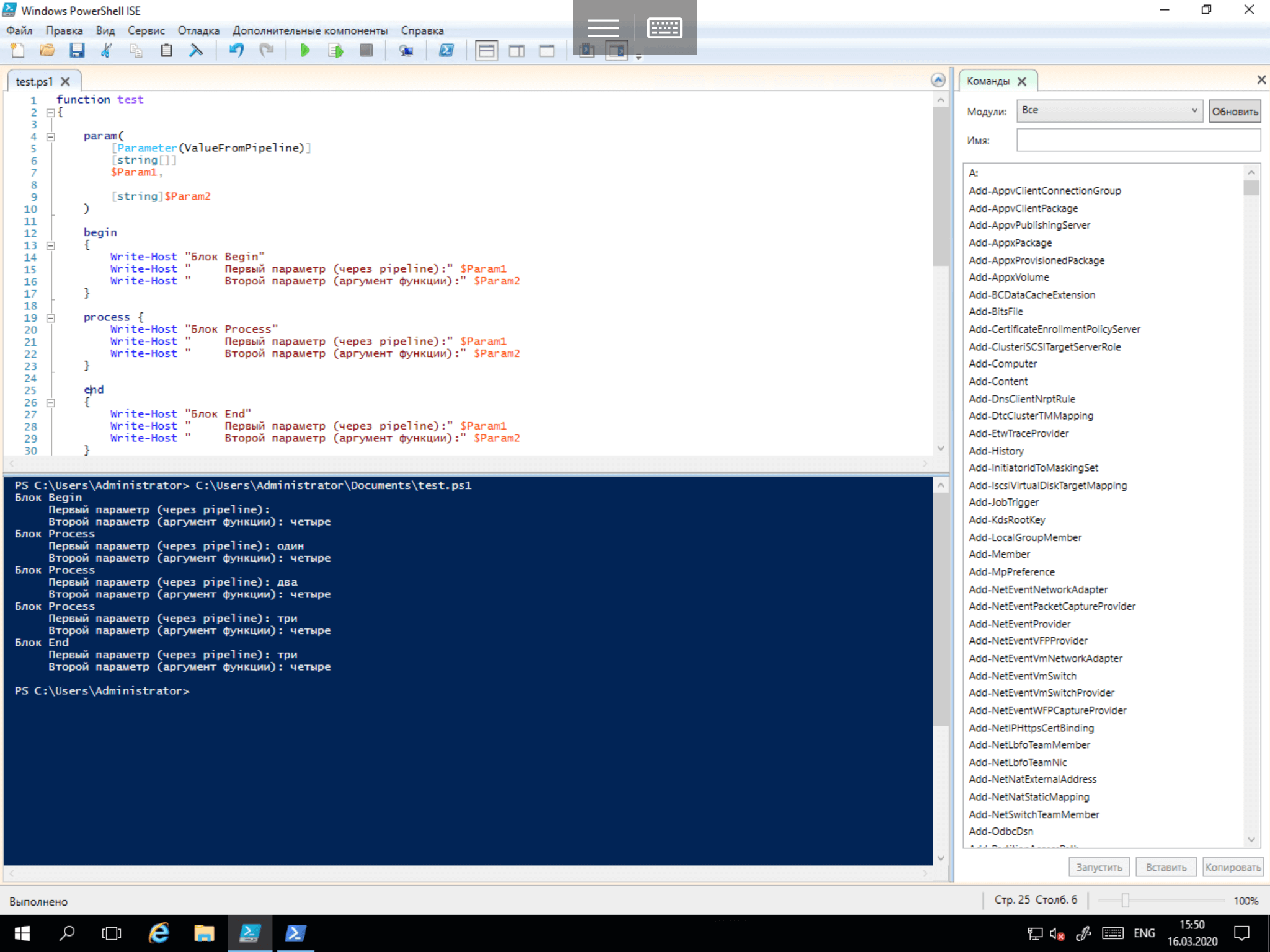
Write-Host "Блок End"

Write-Host " Первый параметр (через pipeline):" $Param1

Write-Host " Второй параметр (аргумент функции):" $Param2

}

}

'один', 'два', 'три' | test -Param2 'четыре'  
  


Атрибут [CmdletBinding()] и расширенные функции

Для создания «продвинутых» функций (и скриптов строго говоря) можно использовать атрибут [[CmdletBinding()](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_functions_cmdletbindingattribute)]. Он, в частности, позволяет определять расширенные функции с возможностями скомпилированных в Visual Studio бинарных командлетов, представляющих собой классы классы .NET Core. Поскольку применяется этот атрибут в основном в функциях, на них мы и остановимся:

function <Name>

{

[CmdletBinding(ConfirmImpact=<String>,

DefaultParameterSetName=<String>,

HelpURI=<URI>,

SupportsPaging=<Boolean>,

SupportsShouldProcess=<Boolean>,

PositionalBinding=<Boolean>)]

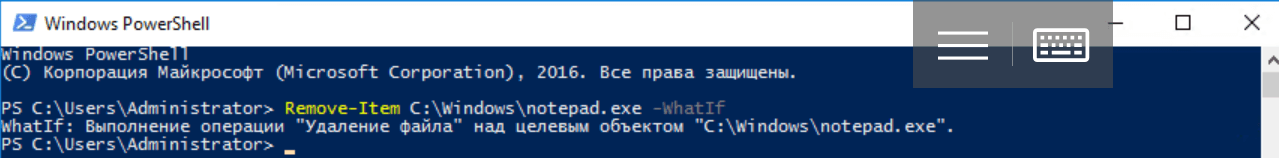
Param ()

Begin{}

Process{}

End{}

}

На самом деле [CmdletBinding()] инициализирует новый экземпляр класса [CmdletBindingAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.management.automation.cmdletbindingattribute) через вызов конструктора, которому можно передать необязательные аргументы. Их подробное описание есть [на сайте](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_functions_cmdletbindingattribute) Microsoft. Атрибут CmdletBinding позволяет контролировать дополнительные возможности расширенной функции: добавление поддержки -Confirm и -WhatIf (через SupportsShouldProcess), -Force, -Verbose и -Debug, а также отключение позиционного связывания параметров и т.д. Дальше мы разберем использование специальных параметров.  
  
Параметр **-Force** применяется для подавления запросов на проведение различных операций;  
  
**-WhatIf** нужен для эмуляции запуска и отображения информации о последствиях выполнения функции (команды) без этого параметра. Обычно используется, если функция может выполнить деструктивные действия.  
  
Remove-Item C:\Windows\notepad.exe -WhatIf  
  
  
  
**-Confirm** требует подтверждения и также используется, если функция может выполнить деструктивные действия.

function Delete-File {

[CmdletBinding(

ConfirmImpact = 'High',

SupportsShouldProcess = $true

)]

param(

[string]$File,

[switch]$Force

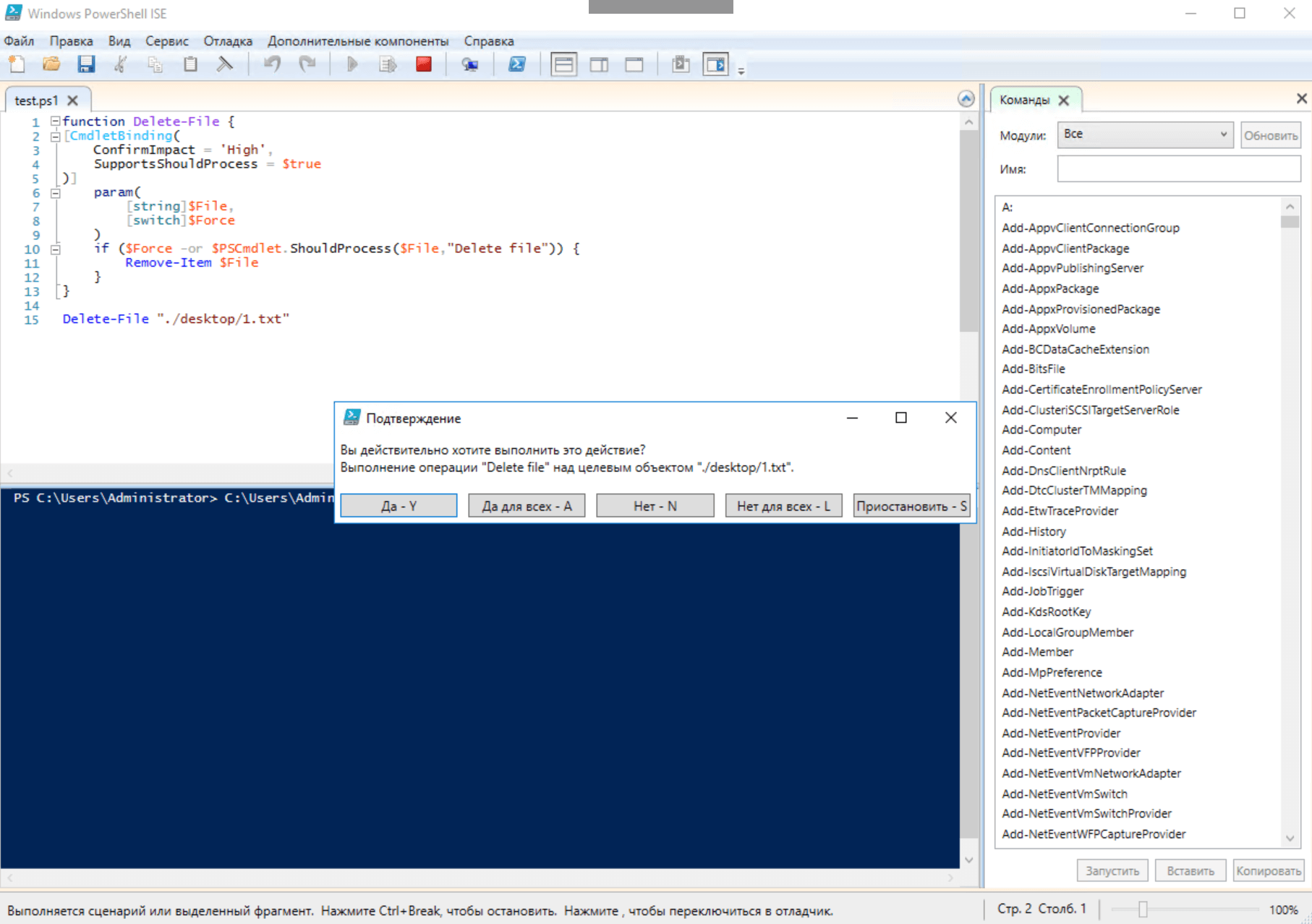
)

if ($Force -or $PSCmdlet.ShouldProcess($File,"Delete file")) {

Remove-Item $File

}

}

  
  
Для обработки -WhatIf и/или -Confirm вызывается метод ShouldProcess (SupportsShouldProcess=$true), который выводит запрос или эмулирует выполнение команды. Чтобы реализовать обработку -Force, мы поместили его первым в условие IF. Сначала проверяется выражение слева от оператора -or и если оно истинно, проверка останавливается — метод ShouldProcess вызываться не будет. Также в атрибуте [CmdletBinding()] мы указали аргумент ConfirmImpact, определяющий уровень воздействия кода на систему и включающий обработчик параметра -Confirm. Этот аргумент может принимать следующие значения:  
  
**None** или не указан — сообщения о подтверждении выводиться не будут, даже если передан параметр -Confirm.  
  
**Low** — функция незначительно воздействует на систему и не создает существенных рисков потери данных.  
  
**Medium** — среднее воздействие с незначительным риском потери данных в результате деструктивных действий.  
  
**High** — код создает высокий риск потери данных в результате деструктивных действий.  
  
По умолчанию для сессии PowerShell уровень воздействия считается высоким (High). Актуальное значение хранится в переменной $ConfirmPreference и если у кода уровень воздействия на систему такой же или более высокий, запрос на подтверждение будет выводиться всегда.  
  
Параметры **-Verbose** и **-Debug** нужны для вывода отладочной информации. Их использование считается хорошим стилем программирования (забудьте про Write-Host, в расширенных функциях это не нужно). Первый параметр выводит информацию о ходе выполнения, а второй — детальную отладочную информацию. Также он дает возможность переключиться на пошаговое выполнение кода. Поведение -Verbose и -Debug определяется примерно так:

function Get-Something {

[CmdletBinding()]

param()

if ($PSBoundParameters.Verbose) {$VerbosePreference = "Continue"}

if ($PSBoundParameters.Debug) {$DebugPreference = "Continue"}

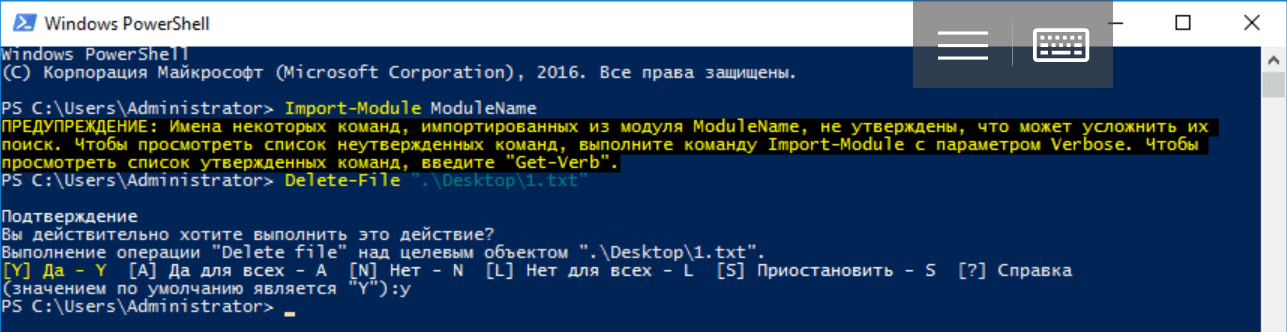
Write-Verbose "Type some verbose information"

Write-Debug "Type some debug information"

}

Для работы со специальными параметрами мы использовали переменную $PSBoundParameters. По умолчанию значения $VerbosePreference и $DebugPreference равны 'SilentlyContinue', поэтому даже при указании соответствующих параметров отладочная информация выводиться не будет — их нужно перевести в состояние 'Continue'.

Модули сценариев и создание командлетов

Приступим к созданию собственных командлетов. По сути это расширенные функции, которые описаны в т.н. [модулях сценариев](https://docs.microsoft.com/ru-ru/powershell/scripting/developer/module/how-to-write-a-powershell-script-module) — текстовых файлах с расширением .psm1. Хранятся они в каталогах, определенных в переменной окружения PSModulePath. Посмотреть пути к ним можно при помощи следующей команды:  
  
Get-ChildItem Env:\PSModulePath | Format-Table -AutoSize  
  
Стандартный набор выглядит примерно так:  
  
C:\Users\%UserName%\Documents\WindowsPowerShell\Modules  
C:\Program Files\WindowsPowerShell\Modules  
C:\Windows\System32\WindowsPowerShell\v1.0\Modules  
  
После создания файла ModuleName.psm1 с расширенной функцией Delete-File из предыдущего раздела, его нужно сохранить, например, в ]C:\Users\%UserName%\Documents\WindowsPowerShell\Modules\ModuleName. Обратите внимание, что модуль сценариев должен храниться в отдельном подкаталоге, имя которого совпадает с базовым именем (без расширения) файла .psm1. После запуска команды **Import-Module ModuleName** функция Delete-File станет доступна пользователю, а поскольку она расширенная, с практической точки зрения это тот же командлет.  
  
  
  
В этой статье мы достаточно подробно разобрали передачу параметров в функции и скрипты. Следующая часть цикла будет посвящена объектно-ориентированному программированию.