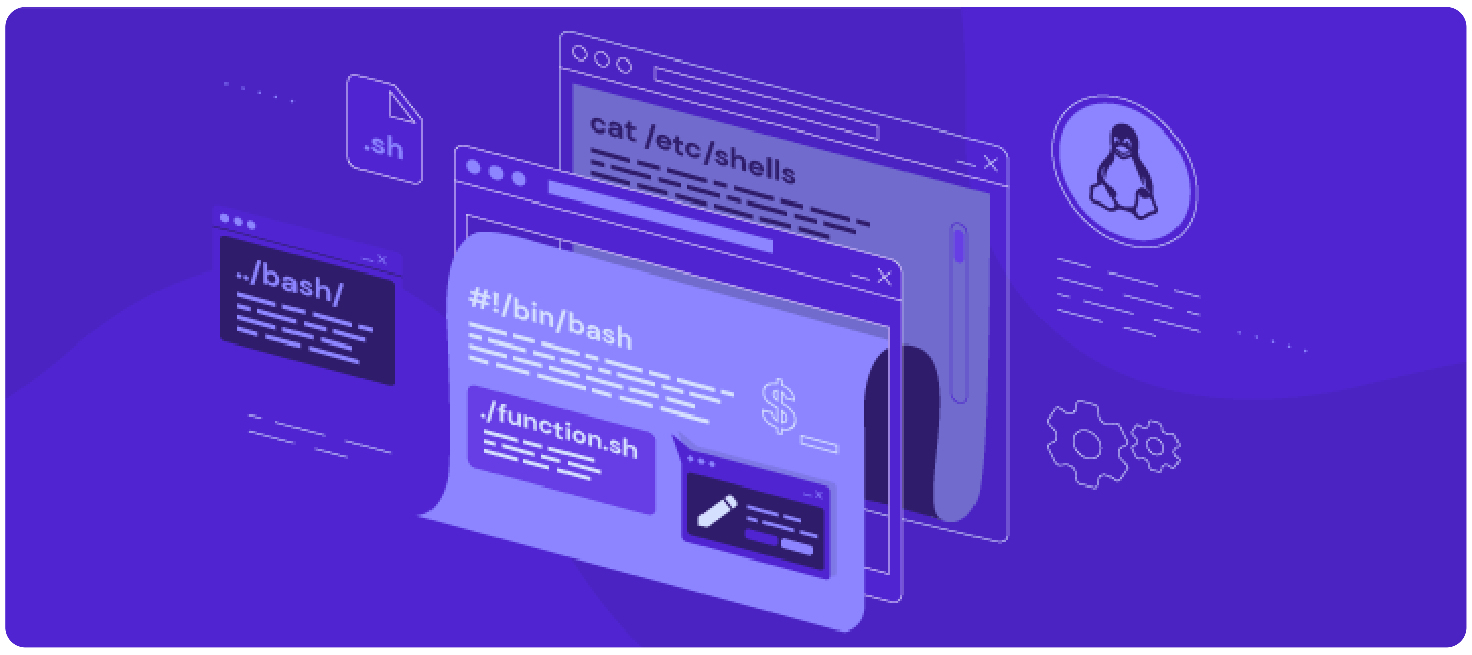
VPS May 25, 2022 Leonardus N. 13min Read

Bash Scripting Tutorial for Beginners: What It Is, How to Write One, and Script Examples



If you've ever used a Linux operating system like most virtual private servers, you may have heard of bash. It's a Unix shell that reads and executes various commands.

When you need to run several bash commands, you don't have to execute them manually one at a time. Instead, it's possible to create a script file that contains bash functions to run those commands.

Download Complete Linux Commands Cheat Sheet

It may sound complicated, but by learning its basics, you will understand the bash scripting language and find out how it can help your workflow.

This article will cover the process of bash scripting. We'll go over everything from bash commands to running a bash program on a Linux terminal.

What Is Bash?

Bash, short for Bourne-Again Shell, is a Unix shell and a command language interpreter. It reads shell commands and interacts with the operating system to execute them.

To fully understand bash shell scripting, you need to know two concepts – shell and scripting.

Meanwhile, scripting is the process of compiling shell commands into a new file using a text editor.

When you write bash in a text editor, you're compiling bash commands or bash functions – a set of commands that can be called numerous times by only using the function name. The text is then saved as an executable bash script file with the .sh extension.

Why Use Bash Scripting?

Bash scripts can help with your workflow as they compile many lengthy commands into a single executable script file.

For example, if you have multiple commands that you have to run at a specific time interval, you can compile a bash script instead of typing and executing the commands manually one by one. You only need to execute the script file when it's necessary.

Here are some other advantages of using bash scripts:

- Well-structured commands – structure the commands in a sequence so that every time you execute the script, it will run in the right order.
- Task automation – automate the script execution at any defined time using cron's time-based scheduler.
- Transparency – people can check the content of a script since it's in a readable text file. However, if you run the commands using another program written in a different programming language, such as C++, you'll need to access the source code.
- Transferable – if you transfer a script to other Linux distributions, it'll still work, providing that shell commands are available on that particular operating system.

Pro Tip

Linux has a bash shell command manual. It contains descriptions of all technical terms and standard shell variables. Type and execute the man bash command to display the manual on the terminal.

Get Familiar With Bash Commands

Bash is available on almost all types of Unix-based operating systems and doesn't require a separate installation. You will need a Linux command line, also known as the Linux terminal. It's a program that contains the shell and lets you execute bash scripts.

Use this command to check the list of available shells on your Unix operating system:

```
cat /etc/shells
```

The output should show a list like this:

```
/bin/bash
/bin/sh
/bin/tcsh
/bin/csh
```

Each bash shell script needs to start with #! followed by the absolute path to the bash interpreter. To view the path, enter this command:

```
which bash
```

It should produce the following output:

This is the standard path to the bash interpreter on most Unix operating systems. To let the shell know that it should run commands using the bash interpreter, start the script with this line:

```
#!/bin/bash
```

Important! If you want to run bash scripts on a virtual private server, connect to it via an SSH client.

The next step is to write and compile the commands in a .sh file using a text editor. You will need a Unix text editor such as VIM or GNU Nano. In this tutorial, we'll use the Nano text editor to create the file by inputting this command:

```
nano function.sh
```

This will open a new .sh file for you to edit. Begin by writing #!/bin/bash followed by bash commands.

Important! Once you're done using the Nano text editor, press Ctrl+X to close it, then press Y and Enter to save the changes.

Basic Bash Commands for Your First Bash Script

In order to successfully create your first bash script, you need to understand the essential bash commands. They are the main elements of a script, and you must know what they do and how to write them properly.

There are a lot of bash commands on Linux. To start things off, we'll cover seven basic ones.

1. Comments

Comments feature a description on certain lines in the script. The terminal doesn't parse comments during execution, so they won't affect the output.

There are two ways to add comments to a script. The first method is by typing # at the beginning of a single-line comment.

```
#!/bin/bash
#Command below prints a Hello World text
echo “Hello, world!”
```

The second method is by using : followed by ‘. This method works for multiple-line comments.

```
#!/bin/bash
read a
: ‘
The following commands prints
Hello, world!
‘
echo “Hello, World!”
```

2. Variables

Variables are symbols that represent a character, strings of characters, or numbers. You only need to type the variable name in a command line to use the defined strings or numbers.

To assign a variable, type the variable name and the string value like here:

In this case, testvar will is the variable name, and This is a test variable is the string value. When assigning a variable, we recommend using a variable name that's easy to remember and represents its value.

To read the variable value in the command line, use the $ symbol before the variable name. Take a look at the example below:

```
#!/bin/bash
testvar="This is a test variable"
echo $testvar
```

The second command line uses echo to print out the value of testvar. The output of that script will be:

```
This is a test variable
```

Let's take a look at how you can enter a string value by using the read command and make the script compare two string values from different variables:

```
#!/bin/bash
echo "Enter a number"
read a #The user input in this command line will be stored as variable a
b=50 #The value of variable b
if [[$a -eq $b]]
then
echo "Same number"
else
echo "Different number"
fi
```

The output of that script should be as follows:

```
Enter a number
20
Different number
```

Note that line 3 is the value that becomes variable a.

The script compares the variable a with the value 20 and the variable b with the value 50. Since the values are different, the script prints out Different number.

However, if the user inputs 50, this will be the output:

```
Enter a number
50
Same number
```

This example also uses conditional statements, which we will discuss later.

3. echo Command

echo is a well-known command used in many programming languages. There are various options you can use with echo to print the output on the terminal.

The first and most common use of echo is to output standard text:

```
#!/bin/bash
echo "Hello, world!"
```

The output of that command is Hello, World! By default, when using the echo command like this, the terminal will input a new line underneath that. If you want to echo an output without a new line, you can do so by using -n.

Use the option \n to introduce a line break into the output. To enable the backslash (\), you need to include -e.

```
#!/bin/bash
echo -e “Hello, \nworld!”
```

The output of that command will look like this:

```
Hello,
world!
```

The option \t adds a horizontal tab space:

```
#!/bin/bash
echo -e “\tHello, world!”
```

This command’s output will indent the text to the right:

```
        Hello, world!
```

You can also combine several options. For example, combine \n and \t to break the text into lines and indent it to the right:

```
#!/bin/bash
echo -e “\n\tHello, \n\tworld!”
```

The output of that command will look like this:

```
      Hello,
      world!
```

4. Functions

A function compiles a set of commands into a group. If you need to execute the command again, simply write the function instead of the whole set of commands.

There are several ways of writing functions.

The first way is by starting with the function name and following it with parentheses and brackets:

```
function_name () {
first command
second command
}
```

Or, if you want to write it in a single line:

```
function_name () { first command; second command; }
```

The second method to write a function is using the reserved word function followed by the function name. This eliminates the need for parentheses:

```
function function_name {
first command
second command
}
```

This method also has a single-line version:

For example, we can write two functions with multiple echo commands:

```
#!/bin/bash
hello_world () {
echo “Hello, World!”
echo “This is a test function”
}
print_message () {
echo “Let’s learn bash programming”
echo “Enjoy this tutorial”
}
```

Note that writing the functions as in the example above only defines them and doesn’t execute the contained commands. To execute a function, enter its name into the command line.

Now, let’s use the two examples above in a complete bash function, including its execution:

```
#!/bin/bash
#Define a hello world function
hello_world () {
echo “Hello, World!”
echo “This is a test function”
}
#Define a print message function
print_message () {
echo “Let’s learn bash programming”
echo “Enjoy this tutorial”
}
#Execute the hello world function
hello_world
#Execute the print message function
print_message
```

This is the output of the script above:

```
Hello, World!
This is a test function
Let’s learn bash programming
Enjoy this tutorial
```

5. Loops

Loop bash commands are useful if you want to execute commands multiple times. There are three types of them you can run in bash – for, while, and until.

The for loop runs the command for a list of items:

```
#!/bin/bash
for item in [list]
do
[commands]
done
```

The following example uses a for loop to print all the days of the week:

```
#!/bin/bash
for days in Monday Tuesday Wednesday Thursday Friday Saturday Sunday
do
echo “Day: $days”
done
```

The output of that script will be as follows:

```
Day: Monday
Day: Tuesday
Day: Wednesday
Day: Thursday
Day: Friday
Day: Saturday
Day: Sunday
```

Notice that even with just one command line in the loop script, it prints out seven echo outputs.

The next type of loop is while. The script will evaluate a condition. If the condition is true, it will keep executing the commands until the output no longer meets the defined condition.

```
#!/bin/bash
while [condition]
do
[commands]
done
```

Let's take a look at a simple example that involves a variable and increment operator, denoted as ((++)):

```
#!/bin/bash
i=0
while [ $i -le 5 ]
do
echo $i
((i++))
done
```

The variable starts with a 0 value, and the increment operator will increase it by one. The condition set is less than or equal to five, so the command will keep iterating until the output reaches five. The output of that script will be as follows:

```
0
1
2
3
4
5
```

The last type of loop, until, is the opposite of while. It will iterate the command until the condition becomes true.

If we want the same output as the while example above using the until loop, we can write the script like this:

```
#!/bin/bash
i=0
until [ $i -gt 5 ]
do
echo $i
((i++))
done
```

Now, this command will iterate until the output value reaches five. The output will be the same as our example with the while loop:

```
3
4
5
```

6. Conditional Statements

Many programming languages, including bash, use conditional statements like if, then, and else for decision-making. They execute commands and print out outputs depending on the conditions.

The if statement is followed by a conditional expression. After that, it's followed by then and the command to define the output of the condition. The script will execute the command if the condition expressed in the if statement is true.

However, if you want to execute a different command if the condition is false, add an else statement to the script and follow it with the command.

Let's take a look at simple if, then, and else statements. Before the statement, we will include a variable so the user can input a value:

```
#!/bin/bash
echo "Enter a number"
read num
if [[$num -gt 10]]
then
echo "The number is greater than 10"
else
echo "The number is not greater than 10"
```

7. Reading and Writing Files

There are several methods to read a file, with the cat command being the most popular one. Note that this command reads the whole file content.

To read the content line by line, use the read command and a loop. Before writing a script to read a file, make sure that the file exists first.

In the following example, we have a to-do.txt file that contains a to-do list:

```
Reply email
Finish report
Call clients
Team evaluation
```

We'll use the cat and read commands in our bash function to read and print the content of the to-do.txt file. The first part of the script will use the cat command, while the second part will use the read command in a loop.

```
#!/bin/bash
echo "Reading the file using cat command"
content='cat to-do.txt'
echo $content
echo "Reading the file using read command and loop"
filename='to-do.txt'
while read line
do
echo $line
done<$filename
```

The output of the script will be as follows:

```
Reply email
Finish report
Call clients
Team evaluation
```

To write a command output into a file, use the redirection operators, represented with the > and >> symbols, and follow them with the file name:

```
output > filename
output >> filename
```

Be careful when choosing the operator. If the file exists, the > operator will overwrite its content with a zero-length string. It means you'll lose the existing file content. If the inputted file name doesn't exist, it will create it.

The >> operator, on the other hand, will add the output to the given file.

Here's a simple redirection to write the output into a text file:

```
echo “Hello, world!” >> hello_world.txt
```

Redirection also works with the read command to write any user input. This script example will add the input value into the name.txt file:

```
#!/bin/bash
echo “Enter your name”
read Name
echo $Name >> name.txt
```

Because the script redirects the variable output into the file, you won't see any output printed. To see the output by printing the file content, add the following command line to read the file:

```
echo ‘cat name.txt’
```

Make sure that you have the permission to read and write in the file to prevent the permission denied error. If you want to add the output to existing files, make sure to type in the correct file names.

4 Easy Functions to Try On Your First Bash Script

Now that we know some bash commands, we'll look at more basic bash function examples for your first script.

As mentioned earlier, when you want to write a bash script file, use the nano filename.sh command to create and open a .sh file and start writing your bash functions. Don't forget to exit and save the file when you're done.

Start with a Simple echo Function

Let's start with a simple echo function. Start by defining the function name followed by the echo command on the next line, just like in the example below:

```
#!/bin/bash
testfunction () {
echo “My first function”
}
testfunction
```

```
My first function
```

Keep in mind that if you swap the position of the function definition with the function call, it will result in an error. Let's see the example below:

```
#!/bin/bash
testfunction
testfunction(){
echo “My first function”
}
```

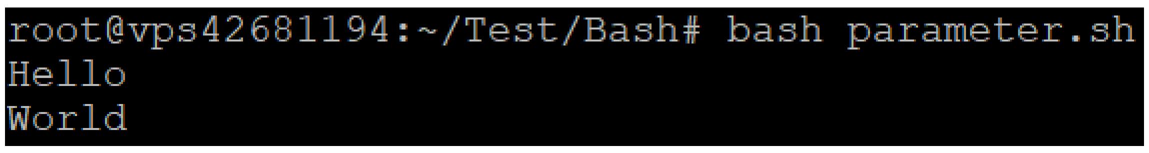
This snippet won't work. It calls the function in the second command line and defines the function later. In this case, the interpreter can't find the function when it executes the script, resulting in a command not found error.

Use a Few Parameters

Bash functions accept any number of parameters. The example below accepts two parameters:

```
#!/bin/bash
testfunction () {
echo $1
echo $2
}
testfunction “Hello” “World”
```

$1 represents the first argument while $2 represents the second argument in the function execution line. As we used "Hello" and "World" for the arguments, the output will look like this:



```
root@vps42681194:~/Test/Bash# bash parameter.sh
Hello
World
```

You can also use the command line arguments and perform bash functions. One such example is shown below:

```
#!/bin/bash
addition () {
sum=$(($1+$2))
return $sum
}
read -p "Enter a number: " int1
read -p "Enter a number: " int2
addition $int1 $int2
echo "The result is : " $?
```

The addition is assigned in a variable sum, and this is returned from the function. Bash functions always return one single value. User input is taken by using read for both numbers. Finally, the result is printed using $? which stores the return value $sum from the function.

Create Directories and Change Paths

Now, let's look at another function example where we first create a directory and then change the path to point to a new location. This function will contain mkdir and cd Linux commands to create a new directory and change the current directory:

```
cd $1
}
sampleFunction myDir
```

Function will read the first argument and create a directory with that name. After you execute the script, check the present working path using the pwd command on the terminal. You'll see that you are currently within the newly created myDir.

Combine Loops and Conditionals

Loops and conditional statements are also popular in bash scripting. We'll look at a few instances of using both in the same script:

```
#!/bin/bash
isvalid=true
count=1
while [ $isvalid ]
do
echo $count
if [ $count -eq 5 ];
then
break
fi
((count++))
done
```

The example above uses while and if statements. This executes the while loop five times after checking the conditional statement.

The output of this script will be:

```
1
2
3
4
5
```

The for loop can increment and decrement the counters. An example of a for loop is shown below:

```
#!/bin/bash
for (( count=10; count>0; count-- ))
do
echo -n "$count "
done
```

The output of this for loop should be:

```
10 9 8 7 6 5 4 3 2 1
```

With if statements, we can also define else if by using elif statement:

```
if [ $n -eq 101 ];
then
echo "This is the first number"
elif [ $n -eq 510 ];
then
echo "This is the second number"
elif [ $n -eq 999 ];
then
echo "This is the third number"
else
echo "No numbers over here"
fi
```

We can also write that script using the case statement. In the case statements, ;; represents a case break, so if the variable value meets any of the conditions, it jumps to the end of the script:

```
#!/bin/bash
echo "Enter a valid number"
read n
case $n in
101)
echo "This is the first number" ;;
510)
echo "This is the second number" ;;
999)
echo "This is the third number" ;;
*)
echo "No numbers over here" ;;
esac
```

How to Run Bash Script

Now that we have written a bash script, let's learn how to run it from the terminal. There are three methods to do it – using the bash command, using the ./ command, and running the script from a different directory.

Using the Bash Command

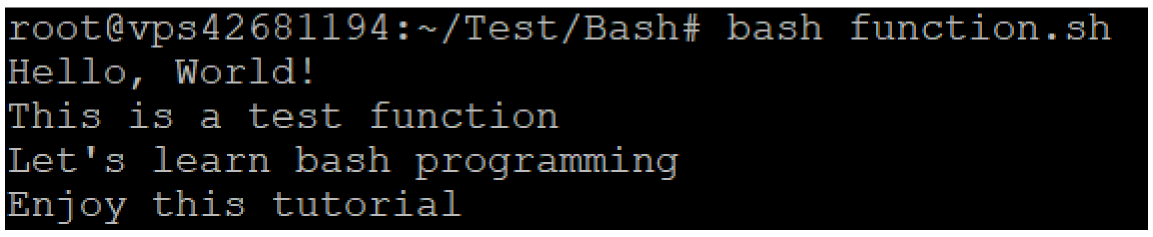
The first method is by using the bash command from the appropriate directory. For example, you may have a function.sh bash script containing simple echo functions in the Test/Bash directory. You have to open the directory first by using this command:

```
cd Test/Bash
```

Then, execute the following bash command to run the bash script:

```
bash function.sh
```

You should see the output like this:



```
root@vps42681194:~/Test/Bash# bash function.sh
Hello, World!
This is a test function
Let's learn bash programming
Enjoy this tutorial
```

If you try to run the script without the bash command, you'll receive a command not found error message.

Using the ./ Command

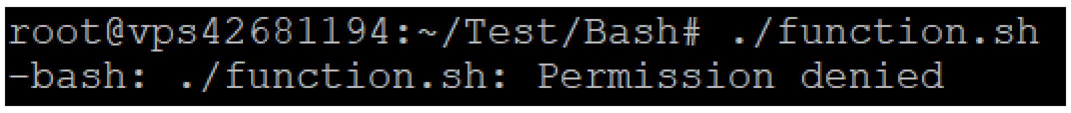
You can run a bash script without the bash command. However, you have to set the file to have the execute permission using the following command from the appropriate directory:

```
chmod +x function.sh
```

This command modifies the file permissions so that everyone can execute the file. Once you've done that, execute the bash script by using this command:

```
./function.sh
```

If you don't set the permission correctly, the terminal will print a Permission denied error message:



```
root@vps42681194:~/Test/Bash# ./function.sh
-bash: ./function.sh: Permission denied
```

Like the bash command, you'll also get a command not found error if you don't use ./ in your command.

Run the Script From a Different Directory

Another alternative is to run the script from a different directory. After you create the bash script, use the pwd command to find your current directory. Once you know the path, you can run the script from any directory. For example, use the following command to run function.sh from the home directory:

```
bash Test/Bash/function.sh
```

Pro Tip

Use the cd command to go to the home directory straight away regardless of the directory you are in.

Conclusion

Bash reads shell commands and interacts with the operating system to execute them. The great aspect of bash is that you can compile many bash commands and functions into a single executable script, helping you streamline your workflow.

To create a bash script file, you need to understand the various shell commands and their proper syntax. In this tutorial, we've covered seven basic commands:

- Comments
- Variables
- Echo
- Functions
- Loops
- Conditional statements
- Reading and writing files

However, there is much more to learn if you want to be able to utilize the full potential of bash. Practice with the examples we have provided and continue exploring bash so you can write better and more efficient scripts.

Looking for more bash guides?

Beginner's Guide to Bash Array

Was this tutorial helpful?

Yes

No

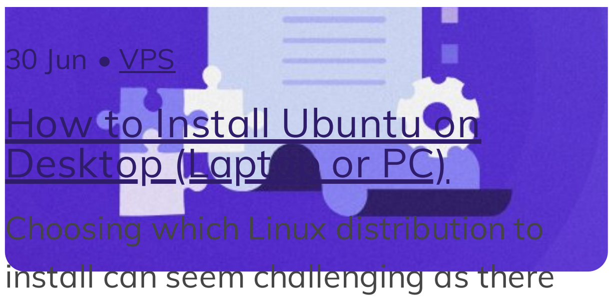
THE AUTHOR

Leonardus N.

Leo is a Digital Content Writer at Hostinger. He loves to share his web hosting and WordPress knowledge to help people build a successful online presence. During his free time, he likes to play music and learn audio engineering.

More from Leonardus N.

Related tutorials



30 Jun • VPS

How to Install Ubuntu on Desktop (Laptop or PC)

Choosing which Linux distribution to install can seem challenging as there are many versions to choose from, and each of them offers very different...

By Ignas Rimkūnas

25 May • VPS

What Is Docker and How Does It Work? – Docker Explained

If you have an application or service and want it to work on different systems like VPSs or dedicated machines without any issues, consider using...

By Ignas Rimkūnas

19 May • VPS

How to Secure Your Linux Server with Fail2Ban Configuration

Fail2Ban is arguably the best software to secure a Linux server and protect it against automated attacks. When enabled, it offers many customizable...

By Noviantika Gita

What our customers say

Excellent

★★★★★

Based on 9,321 reviews

Trustpilot

Leave a reply

Comment*

Name*

Email*

☐ By using this form you agree that your personal data would be processed in accordance with our Privacy Policy.

Submit

HOSTINGER TUTORIALS

Cart

And More

HOSTING

Web Hosting
VPS Hosting
Minecraft Server Hosting
CyberPanel Hosting
Cloud Hosting
WordPress Hosting
Email Hosting
CMS Hosting
Ecommerce Hosting
Free Website Hosting
Online Store
Website Builder
Buy Hosting
Hosting for Agencies

DOMAINS

Domain Checker
Domain Transfer
Free Domain
XYZ Domain
Cheap Domain Names
Domain Extensions
WHOIS Lookup
Free SSL Certificate

HELP

Tutorials
Knowledge Base
Report Abuse

INFORMATION

Migrate to Hostinger
System Status
Affiliate Program
Payment Methods
Wall of Fame
Reviews
Pricing
Sitemap

COMPANY

About Hostinger
Our Technology
Career
Newsroom
Contact Us
Blog
Student Discount

LEGAL

Privacy Policy
Terms of Service

© 2004-2022 hostinger.com - Premium Web Hosting, Cloud, VPS & Domain Registration Services. Prices are listed without VAT