

# Учебная практика BASH-скриптов для чайников

## От автора

Данная методическое пособие написано для потока 1-го курса ИУ7 2023 года. Автор не гарантирует истинность изложенного ниже. Всё, что написано, основано на опыте автора в создании скриптов на языке BASH.

Автор предполагает, что читатель освоил основы языка программирования Python.

Замечания, предложения отправлять:

- В telegram: [@zhikhkirill](#)
- В VK: [@zhikh.localhost](#)

## Глава I. Да кто этот ваш `.bash`?

Программисты — люди ленивые, привыкшие все автоматизировать. Поэтому неудивительно, что появилась такая вещь, как bash-скрипты (далее автор будет называть bash-скрипты просто **bash**). В bash можно записывать целые сценарии для выполнения рутинной работы. Приведу пример:

```
#!/bin/bash
# test.sh

for file in "./tests/in/*"; do
    cat $file | python3 my_lab.py > ${file/in/out}
done
```

Скрипт автоматически вводит данные в Вашу программу (например, лабораторную по программированию) и записывает результаты в файл. Таким образом Вы можете протестировать свою программу сразу на нескольких массивах входных данных *одной командой!*

```
bash test.sh
```

Неплохо, не правда ли? А ведь это только начало! Возможности BASH-скриптов ограничены лишь Вашим воображением.

И да, `bash` есть на `Linux`-системах. Предмет ПТП подразумевает, что у Вас уже есть данная ОС в любом виде.

Перейдём к основному материалу.

## Глава II. Строки — это всё

В bash-е нет типов переменных!

Просто запомните это. Каждый раз, когда Вы используете переменную, помните эти слова. Чтобы лучше это понять, я задам Вам вопрос: а есть ли типы данных в языке программирования Python (далее — просто Python)? Ведь мы можем сделать так:

```
a = 5
a = "foo"
```

Как будто у переменной `a` нет определённого типа. Однако это не означает, что в Python нет типов данных. Просто *переменная не имеет строго определённого типа*, т.к. в этом ЯП (язык программирования) *динамическая типизация*.

```
>>> a = "foo"
>>> type(a)
<class 'str'>
>>> a = 5
>>> type(a)
<class 'int'>
```

А вот с bash-ем все не так:

В bash все переменные являются строками!

Исходя из этого утверждения будет строиться вся логика работы bash-a.

## 2.1. Объявление переменных

Все просто, например переменная `var` со значением 5:

```
var=5
```

Обязательно без пробелов!

```
var=5      # CORRECT
var = 5    # INCORRECT
var= foo   # INCORRECT
var =bar   # INCORRECT
```

## 2.2. Чтение переменных или почему программистам так много платят

Просто потому, что они используют символ доллара `$` для чтения переменных:

```
a=5
echo $a      # 5
```

Иногда нужно чётко ограничить название переменной, например:

```
a=foo
abar="i wanted something else..."
echo $abar      # i wanted something else...
echo ${a}bar    # foobar
```

Как вывести знак доллара, если он означает чтение переменной? Используем экранирование:

```
a=5
echo $a
echo \$a
```

Вывод:

```
5
$a
```

## 2.3. Ввод переменных пользователем

Если переменные вводятся построчно:

```
read a
read b
```

Если через пробел:

```
read a b
```

Обратите внимание: никаких `$` не нужно!

Если нужно ввести *не* через пробел, а, например, через запятую и пробел, можно изменить разделитель:

```
IFS=", "
```

Хорошей практикой будет сохранять значение предыдущего разделителя:

```
OLD_IFS=$IFS
IFS=","

read a b

IFS=$OLD_IFS
```

## 2.4. Кавычки? Не, не слышали

Автор часто замечал такую запись:

```
# WRONG
a="foo"
```

Или:

```
# WRONG
echo "${a}"
```

Однако, кавычки в этих случаях *избыточны*. Нам ничто не мешает сделать так:

```
# CORRECT
a=foo
echo $b
```

Потому что вне зависимости от наличия кавычек, переменная все равно будет, и будет она *строковой*.

Но если Вам нужно создать переменную с пробелом внутри, то кавычки обязательны!

```
a="foo bar"
```

Аналогично для вывода данных через `echo`. Кавычки в большинстве случаев можно опускать:

```
a=foo
b=bar
echo "$a $b"
echo $a $b
```

Вывод будет:

```
foo bar  
foo bar
```

А вот случай, когда без кавычек никак:

```
echo "$a "
```

А как вывести строку с кавычками? Применяем экранирование:

```
a=foo  
b=bar  
echo "\"$a $b\""
```

Тогда вывод будет:

```
"foo bar"
```

Не нужен перевод на новую строку? Делаем так:

```
echo -n "some"  
echo " text"
```

Вывод:

```
some text
```

## 2.5. Позиционные переменные

Страшное название? Но крайне полезная вещь. Вот пример:

```
# test.sh  
echo \"$0=$0"  
echo \"$1=$1"
```

Запускаем вот так:

```
your-directory@username$ bash test.sh hello
```

Получаем ответ:

```
$0=your-directory/test.sh
$1=hello
```

Как Вы заметили, `$n`, где `n` — неотрицательное число, это ничто иное, как аргументы при запуске скрипта. Причём первый (т.е. нулевой аргумент) *это всегда путь до исполняемого файла*. Т.е. если ваша программа лежит тут:

```
/home/iushnik-s-semerki/scripts/my_script.sh
```

То переменная `$0` соответственно будет равна этому значению.

А для остальных:

```
# test.sh
echo \ $1=$1
echo \ $2=$2
echo \ $3=$3
```

```
bash test.sh foo bar "foo bar"
```

```
$1=foo
$2=bar
$3=foo bar
```

Ещё один случай применения кавычек.

Возможно, у Вас возник вопрос: что, если мы не зададим аргумент при вызове скрипта? Что происходит, если переменная не существует?

И тут мы подходим к следующему важному пункту.

## 2.6. Вопросы существования с точки зрения программиста

Итак, попробуем:

```
a=5
echo $a
echo $b
```

И мы получим...

```
5
```

... просто пустую строку! Никаких вам `NameError: name 'b' is not exists!`

В bash несуществующая переменная является пустой строкой

Поэтому код ниже абсолютно рабочий:

```
echo $a          # Empty output
b=$a             # Empty var 'b'
c="my answer: $a" # my answer:
```

Ещё можно удалить переменную командой `unset`

```
a=5
echo $a      # 5
unset a
echo $a      # nothing
```

А как определять, существует ли переменная? Об этом пойдёт речь в следующей главе.

## Глава III. Ветвления в bash-е или "А что если..."

### 3.1. Синтаксис

Красиво и понятно. Что ещё можно сказать?

```
if [ $a ]
then
    echo Hello!
fi
```

Несколько логических строчек кода можно расположить на одной физической, применяя разделитель `;`:

```
if [ $a ]; then
    echo Hello!
fi
```

и мы получим классическую запись условия.

Автор мог бы подробнее разобрать, что такое `[ ]` на самом деле, чем отличается от `[[ ]]` и т.п... Но на самом деле просто пишите `[ ]`, если уверены, что для условия внутри это сработает и `[[ ]]`, если не уверены.

Немного вариативности в условиях:

```
if [ $a ]; then
    echo $a
else
    echo ERROR
fi
```

Или так:

```
if [ $b ]; then
    echo $a
else if [ $b ]; then
    echo $b
fi
fi
```

Обратите внимание: каждый раз, открывая условие `if`, необходимо его закрыть `fi`. Дважды использовали `if` — дважды закрыли `fi`.

### 3.2. Социальное дистанционирование `[ ]`

Важное замечание: `[ ]` *требуют пробелов рядом с собой!*

```
if [$a]; then echo foo; fi # INCORRECT
if [ $a ]; then echo bar; fi # CORRECT
```

Единственное исключение, как Вы догадались:

```
if [[ condition ]]; then echo foo; fi
```

`[[ ]]` записываются без пробела друг между другом.

### 3.3. Вопросы бытия, часть II

А теперь ответим на вопрос, заданный в предыдущей главе. Проверить переменную на существование очень просто:



```
if [ $a ]; then
    echo $a
else
    echo "\$a is not exists"
fi
```

Просто и надёжно. Пользуйтесь на здоровье!

### 3.4. Advanced-level ветвлений

#### Логические операторы и эквивалентность:

В порядке приоритета:

1. не = !
2. и = &&
3. или = ||

Пример использования:

```
if [ ! $a ] && ! [ $b ] || [ $c ]; then
    ...
fi
```

Т.е. ! можно записать как внутри [ ], так и снаружи, чего не скажешь про && и ||.

Можно использовать ( ) для изменения приоритета:

```
if ! ([ $a ] && [ $b ]); then
    echo ERROR
fi
```

== и !=

Несложно догадаться, для чего нужны эти операторы. Ставить внутри скобок:

```
if [[ $a == $b ]]; then
    echo \'a\' is \'b\'
fi
```

```
if [[ $a != $b ]]; then
    echo \'a\' is not \'b\'
fi
```

Тот случай, когда нужно использовать `[[ ]]`. Просто запомните.

### 3.5. Жизни пробелов тоже важны!

Пробелы очень важны! Очень легко потерять пробел и сломать скрипт:

```
! [ $a ]           # INCORRECT
! ( [ $a ] && [ $b ] ) # INCORRECT
[ $a==$b ]         # INCORRECT
[ $a!= $b ]        # INCORRECT
```

Конечно, есть исключения. Но в таких случаях лучше используйте пробелы.

### 3.6. Загадочные минусы

Есть более архаичная запись всего того, что было перечислено:

```
[ $a -a $b ]      # AND
[ $a -o $b ]      # OR
[ $a -eq $b ]     # ==
```

Но тут придётся использовать экранирование `()`:

```
[ ! \( $a -a $b \) ]      # equivalent to "! ([ $a ] && [ $b ])"
```

Смысл использовать старую запись условий, конечно, есть... Но необходимости в этом нет. Можете смело применять синтаксис, рассмотренный выше.

Забегая вперёд: если `$a` и `$b` являются числами, можно использовать такие операторы:

```
[ $a -lt $b ]      # <
[ $a -ge $b ]      # >=
```

Но на этом автор оставит минусы в покое и далее использоваться они не будут.

### 3.7. `=~`

Оператор `=~` достоин отдельного пункта. Используется он только в двойных скобках `[[ ]]`, и определяет соответствие строки регулярному выражению.

Синтаксис:

```
if [[ $a =~ ^[+-]?[0-9]+$ ]]; then
    echo \"\$a\" is number!
fi
```

Регулярное выражение записывается без ". Эти символы будут восприняты буквально, и Ваша регулярка скорее всего будет работать не так, как вы хотели.

Что такое "регулярное выражение"? А об этом пойдёт речь в следующей главе...

### 3.8. Короткая запись условия

...но прежде чем мы перейдем к следующему разделу, для тех, кто честно дочитал до этого момента (да ведь?), будет показан интересный трюк.

Вместо того, чтобы писать

```
if [ ! $a ]; then
    echo ERROR
fi
```

Можно сделать так:

```
[ ! $a ] && echo ERROR
```

Или даже так:

```
[ $a ] && echo $a || echo ERROR
```

Что будет эквивалентно:

```
if [ $a ]; then
    echo $a
else
    echo ERROR
fi
```

Красиво, не правда ли? Но

Используйте данный метод с большой осторожностью! А лучше и вовсе избегать его!

Почему с осторожностью? Попробуйте самостоятельно разобраться, как на самом деле работает данное выражение (зная, что на самом деле значат [ ]), и сделайте вывод.

Почему избегать? Потому что злоупотребление такой формой приводит к нечитабельности кода. Автор обычно использует короткую форму записи условия для проверки чего-либо в командной оболочке, когда нужно выполнить буквально пару команд, например:

```
[ $a ] && echo a exists
```

## Глава IV. Что такое регулярные выражения и с чем их есть?

<https://regex101.com> — регулярные выражения лучше всего писать и тестировать именно здесь. И все примеры лучше проверять здесь

Регулярное выражение, простыми словами — некоторый шаблон, по которому можно определить соответствие строк. Это не строгое сравнение, как в случае с `==`, а "умное". По регулярным выражениям есть целый предмет — Теория Формальных Языков (ТФЯ), кошмар для ИУ9.

Регулярные выражения — очень полезный инструмент, с базовыми основами которого попробуем разобраться в этой главе.

*Единственное, что не могут регулярные выражения — так это [распарсить HTML](#)*

### 4.1. Это база!

Регулярные выражения могут не использовать специальных символов:

```
text
```

Для строки:

```
some text
```

данному регулярному выражению будет соответствовать подстрока:

```
text
```

(проверьте на сайте выше, как это выглядит)

Т.е. если мы сделаем так:

```
if [[ "some text" =~ text ]]; then
    echo yes
else
    echo no
fi
```

Мы получим ответ... yes, т.к. есть подстрока, которая соответствует регулярному выражению.

Далее будут показаны спец. символы. Любой спец. символ может обозначать сам себя при помощи экранирования \:

#### 4.2. Снова [ ]

[ ] - обозначают любой символ внутри. Например:

```
[ab]c
```

Будут соответствовать строки:

```
ac  
bc
```

Чтобы не писать весь алфавит, можно использовать диапазоны:

- [A-Z] - заглавные латинские буквы
- [a-z] - строчные латинские буквы
- [A-z] - латинские буквы (именно в таком порядке!)
- [0-9] - цифры
- [0-f] - так тоже можно, это будут шестнадцатеричные цифры в нижнем регистре

Например:

```
[0-2][0-9]
```

Будут соответствовать строки:

```
00  
18  
29
```

и т.п.

Чтобы не перебирать все возможные символы, "любой символ" можно задать .:

```
..
```

Будут соответствовать строки:

```
ab  
5$
```

## Экранирование

Если нужна точка, как символ точки, используйте экранирование:

```
\. .
```

Соответствует:

```
.a  
.5
```

## 4.3. Кванторы

Если простым языком, кванторы определяют количество предыдущей строки/символа в подстроке.

- `?` - от 0 до 1

```
[0]?[1-9]
```

Будут соответствовать строки:

```
2  
01
```

- `*` - больше 0

```
[0-9]*
```

Будут соответствовать строки:

```
0  
35434646
```

- `+` - больше 1

```
[0-9]+
```

Будут соответствовать строки:

```
1  
1235
```

## 4.4. Повторения

Если нужно конкретное количество повторений подстроки, используются такие скобки — `{}`:

```
[1-9][0-9]{3}
```

Будут соответствовать строки:

```
1830  
2023
```

Или можно записать так:

```
[1-9][0-9]{1-3}
```

Будут соответствовать строки:

```
10  
128  
1830
```

## 4.5. Группы

Если кванторы нужно применить не к одному символу, можно оборачивать нужную подстроку в *группу* `( )`:

```
(ab)*
```

Будут соответствовать строки:

```
ab
ababab
```

## 4.6. Якоря

Якоря обозначают начало и конец строки:

- `^` - начало строки
- `$` - конец строки

При помощи якорей можно "отбросить" соответствие на подстроки. Например:

```
[[ "file.txt.a" =~ .*\.txt ]]      # True
[[ "file.txt.a" =~ .*\.txt$ ]]    # False
[[ "file.txt" =~ .*\.txt$ ]]      # True
```

## 4.7. Числа!

Итак, наконец мы получили достаточно знаний для написания регулярного выражения для целого числа:

```
^[+-]?[0-9]+$
```

Разберем каждый символ:

- `^` - начало строки, т.е. мы отбрасываем "ab566" значения
- `$` - конец строки, т.е. мы отбрасываем "1223ab" значения
- `[+-]` - любой символ, + или -
- `[+-]?` - обозначет, что "+" или "-" необязательны, но они могут быть в строке, обозначающей число
- `[0-9]` - любая цифра
- `[0-9]+` - цифра должна быть хотя бы одна

Можно попробовать сделать регулярное выражение для дробного числа:

```
^[+-]?[0-9]+(\.[0-9]+)?$
```

Подробнее:

- `[+-]?[0-9]+` - проверка на подстроку-число из предыдущего примера. Обратите внимание, что якоря "переехали", т.к. начало и конец строки теперь в других местах.
- `\.` - точка как символ точки
- `\.[0-9]+` - дробная часть числа



- `(\.[0-9]+)?` - дробная часть числа необязательная

Автор настоятельно рекомендует самостоятельно попробовать написать регулярные выражения для даты, для закрепления материала.

## Глава V. Магия математики

Только теперь можно приступить к математическим операциям. Числа желательно проверять регулярным выражением, т.к. все в bash-у является строкой и нет отдельного типа данных для чисел.

```
read a b
re_num=^[ -+]?[0-9]+$
if ! ([ $a =~ re_num ] && [ $b =~ re_num ]); then
    echo ERROR
else
    ...
fi
```

### 5.1. Школьная арифметика

Bash не поддерживает арифметические операции с дробными числами.

Совсем. И незачем. Bash-у не нужно что-то считать, с этой задачей справится любой классический полноценный язык программирования. Поэтому берём только целые числа.

Для произведения *любых* математических операций используются скобки `$(( ))`:

```
a=5
b=6
echo $((a + b))
```

Как можно заметить, `$` внутри математических скобок писать *необязательно*.

Какие операции поддерживаются?

- `+`, `-`, `*`, `/` (целочисленное), `%`, `**`
- `+=`, `-=`, `*=` и т.п.
- `++` и `--`

### 5.2. Сложение строк?

Сложить строки в bash как в Python-е нельзя. Любые попытки подставить *не число* в `$(( ))` окончится провалом. И просто `+` тоже записать нельзя. "А как тогда сложить строки?". Очень просто. Конкатенация строк (а именно так это называется) выглядит так:

```
a=ab
b=5
```

```
c=${a}${b} # ab5
```