BINF2111 – Introduction to Bioinformatics Computing

BASH 101 – while wild loops of function

Richard Allen White III, PhD
RAW Lab

Lecture 11 – Tuesday Sep 24th, 2024

Learning Objectives

- Review quiz/bonus

- Review lab 4

- Review bash for loops

- Bash while loops

- Bash functions

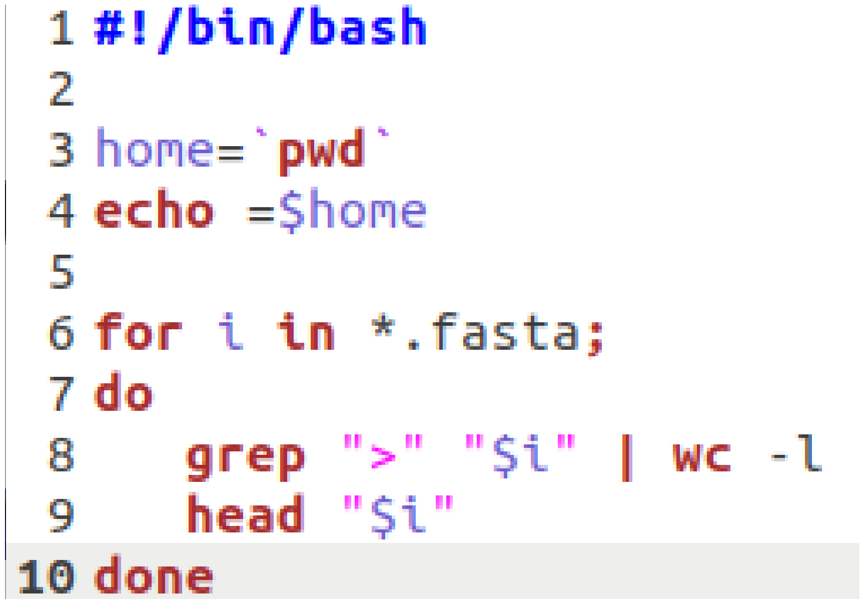
- Quiz 11

Bonus 9

- Write a bash script that prints the working directory, counts all the sequences within a fasta files within the working directory, and prints the first five lines of the file into std_out.txt?

Bonus 9

- Write a bash script that prints the working directory, counts all the sequences within a fasta files within the working directory, and prints the first five lines of the file into std_out.txt?



```
#!/bin/bash

home=`pwd`
echo =$home

for i in *.fasta;
do
    grep ">" "$i" | wc -l
    head "$i"
done
```

Quiz 10

My input is:

```
more file.tsv

bill rod david
Xi abdul larry

perl -pi -e 's/\t/,/' file.tsv

more file.tsv
```

My output is:

```
bill,rod,david
Xi,abdul,larry
```

True or False?

Quiz 10

My input is:

```
more file.tsv

bill rod david
Xi abdul larry

perl -pi -e 's/\t/,/' file.tsv

more file.tsv
```

My output is:

```
bill,rod,david
Xi,abdul,larry
```

True or False

Quiz 10

My input is:

```
more file.tsv

bill rod david
Xi abdul larry

perl -pi -e 's/\t/,/' file.tsv

more file.tsv
```

My output is:

```
bill,rod,david
Xi,abdul,larry
```

```
perl -pi -e 's/\t/,/' file.tsv

more file.tsv
bill	rod,david
xi	abdul,larry
```

Quiz 10

My input is:

```
more file.tsv

bill rod david
Xi abdul larry

perl -pi -e 's/\t/,/' file.tsv

more file.tsv
```

My output is:

```
bill,rod,david
Xi,abdul,larry
```

```
perl -pi -e 's/\t/,/' file.tsv

more file.tsv
bill  rod,david
xi    abdul,larry
```

How do I convert all the way?

Quiz 10

My input is:

```
more file.tsv

bill rod david
Xi abdul larry

perl -pi -e 's/\t/,/' file.tsv

more file.tsv
```

My output is:

```
bill,rod,david
Xi,abdul,larry
```

```
perl -pi -e 's/\t/,/g' file.tsv

more file.tsv
bill  rod david
xi    abdul   larry
```

Does this work?

perl -pi -e 's/\t/,/g' file.tsv >test.csv

Quiz 10

My input is:

```
more file.tsv

bill rod david
Xi abdul larry

perl -pi -e 's/\t/,/' file.tsv

more file.tsv
```

My output is:

```
bill,rod,david
Xi,abdul,larry
```

```
perl -pi -e 's/\t/,/g' file.tsv

more file.tsv
bill  rod david
xi    abdul   larry
```

Does this work?

perl -pi -e 's/\t/,/g' file.tsv >test.csv

NO?

Quiz 10

My input is:

```
more file.tsv

bill rod david
Xi abdul larry
```

```
perl -pi -e 's/\t/,/' file.tsv
```

```
more file.tsv
```

My output is:

```
bill,rod,david
Xi,abdul,larry
```

```
perl -pi -e 's/\t/,/g' file.tsv
```

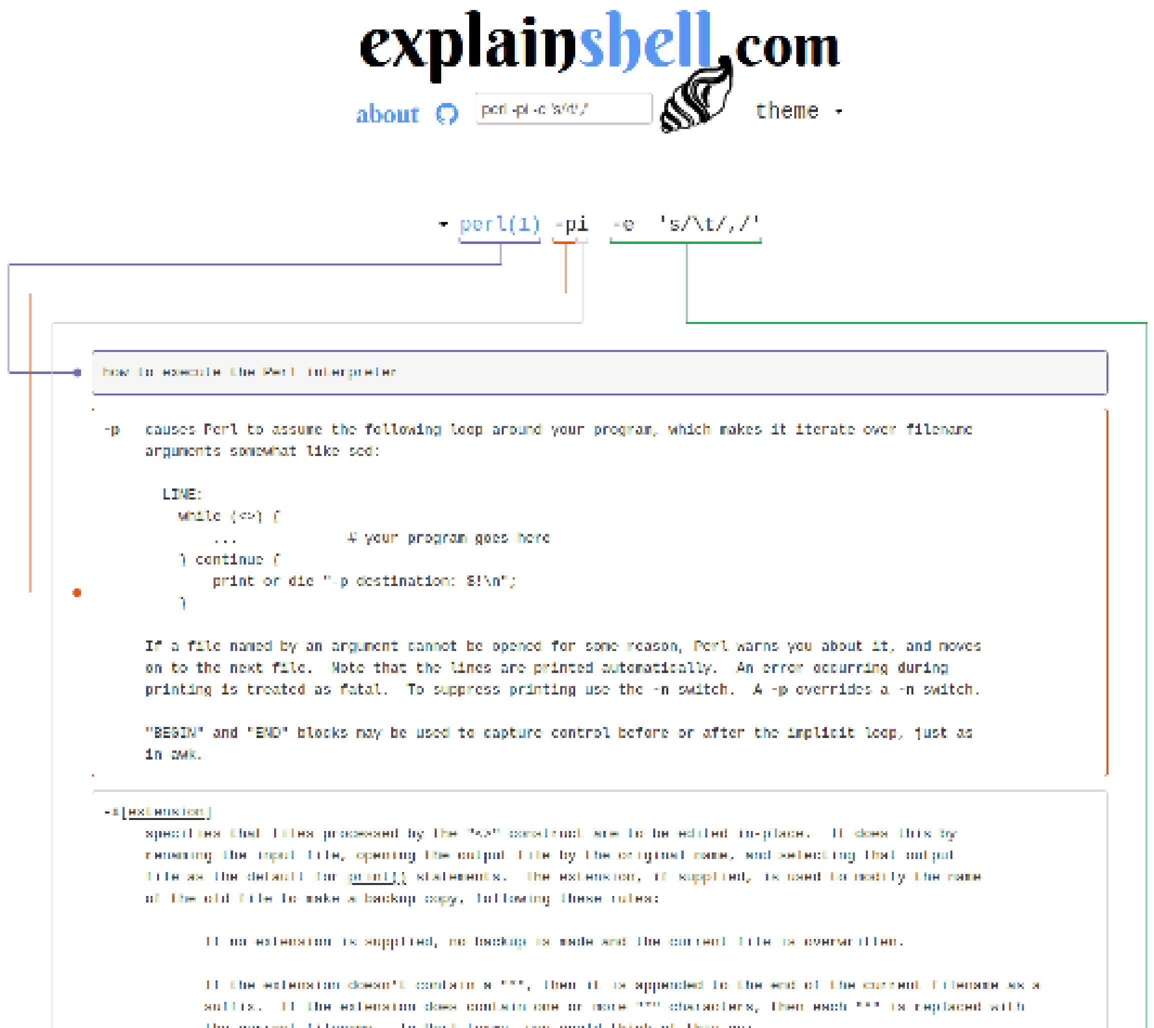
```
more file.tsv
bill  rod david
xi    abdul   larry
```

Does this work?

`perl -p -e 's/\t/,/g' file.tsv >test.csv`

Remove the "i"

Quiz 10



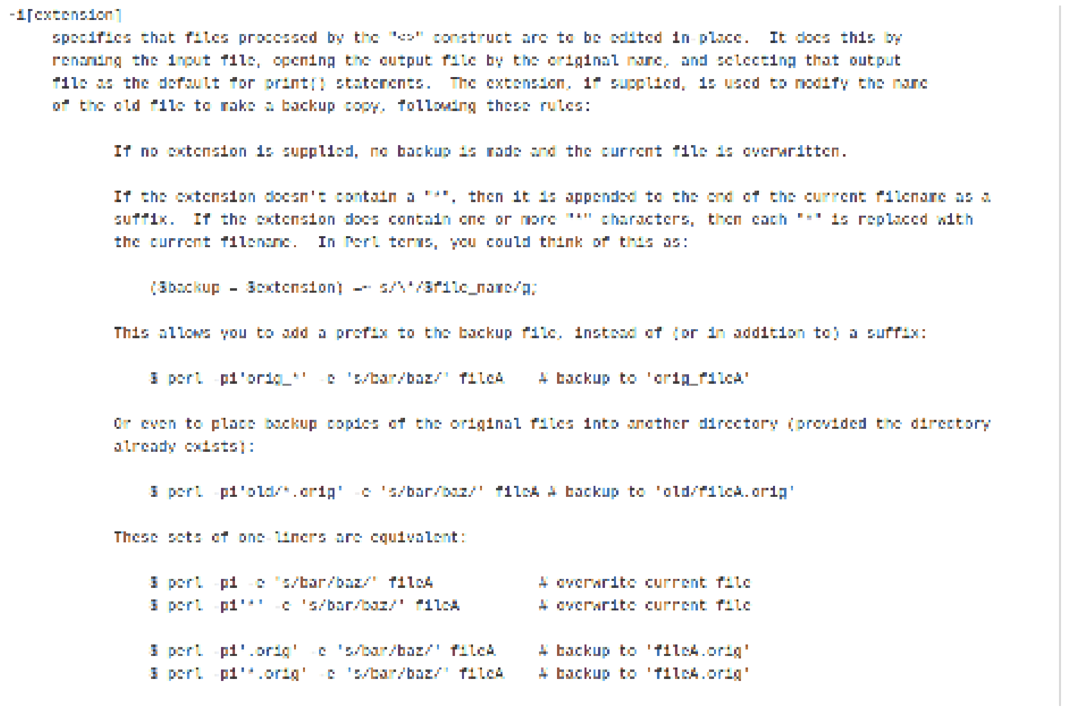
```
perl(1) -pi -e 's/\t/,/'
```

how to execute the Perl interpreter

```
-p   causes Perl to assume the following loop around your program, which makes it iterate over filename
     arguments somewhat like sed:

       LINE:
         while (<>) {
             ...             # your program goes here
         } continue {
             print or die "-p destination: $!\n";
         }

     If a file named by an argument cannot be opened for some reason, Perl warns you about it, and moves
     on to the next file.  Note that the lines are printed automatically.  An error occurring during
     printing is treated as fatal.  To suppress printing use the -n switch.  A -p overrides a -n switch.

     "BEGIN" and "END" blocks may be used to capture control before or after the implicit loop, just as
     in awk.
```

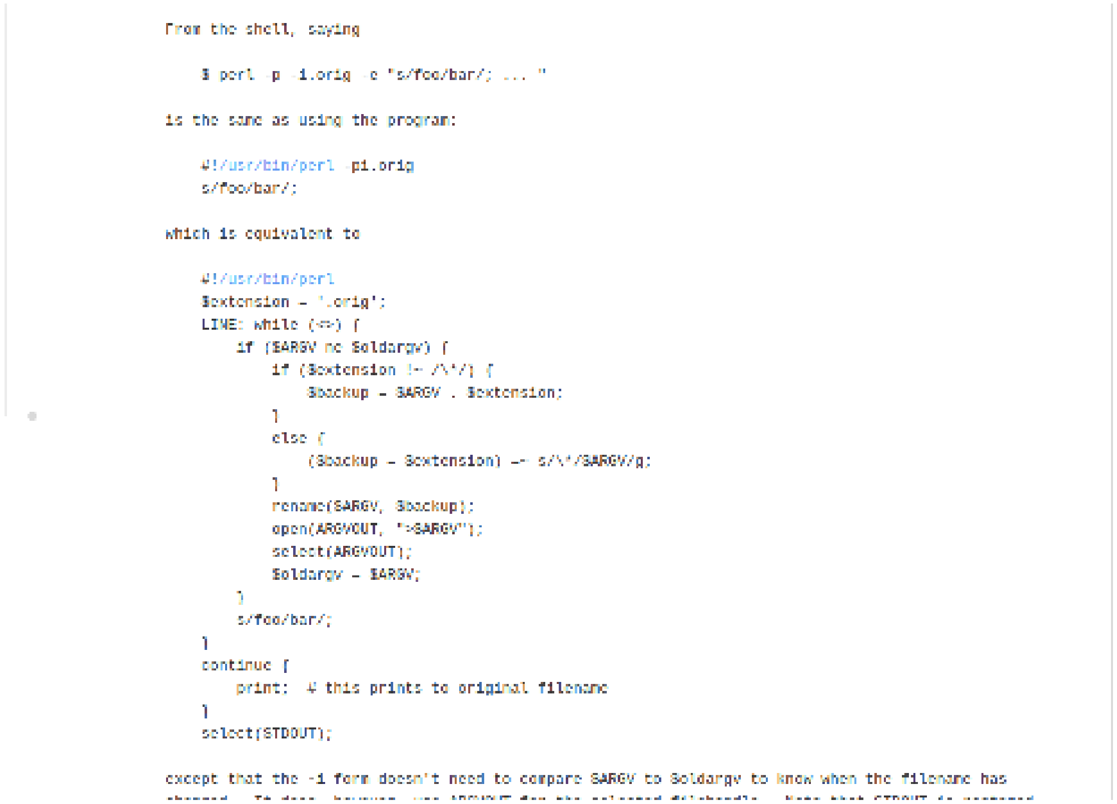
```
-i[extension]
     specifies that files processed by the "<>" construct are to be edited in-place.  It does this by
     renaming the input file, opening the output file by the original name, and selecting that output
     file as the default for print() statements.  The extension, if supplied, is used to modify the name
     of the old file to make a backup copy, following these rules:

         If no extension is supplied, no backup is made and the current file is overwritten.

         If the extension doesn't contain a "*", then it is appended to the end of the current filename as a
         suffix.  If the extension does contain one or more "*" characters, then each "*" is replaced with
```

Quiz 10



```
-i[extension]
     specifies that files processed by the "<>" construct are to be edited in-place.  It does this by
     renaming the input file, opening the output file by the original name, and selecting that output
     file as the default for print() statements.  The extension, if supplied, is used to modify the name
     of the old file to make a backup copy, following these rules:

            If no extension is supplied, no backup is made and the current file is overwritten.

            If the extension doesn't contain a "*", then it is appended to the end of the current filename as a
            suffix.  If the extension does contain one or more "*" characters, then each "*" is replaced with
            the current filename.  In Perl terms, you could think of this as:

                ($backup = $extension) =~ s/\*/$file_name/g;

            This allows you to add a prefix to the backup file, instead of (or in addition to) a suffix:

                $ perl -pi'orig_*' -e 's/bar/baz/' fileA    # backup to 'orig_fileA'

            Or even to place backup copies of the original files into another directory (provided the directory
            already exists):

                $ perl -pi'old/*.orig' -e 's/bar/baz/' fileA # backup to 'old/fileA.orig'

            These sets of one-liners are equivalent:

                $ perl -pi -e 's/bar/baz/' fileA            # overwrite current file
                $ perl -pi'*' -e 's/bar/baz/' fileA         # overwrite current file

                $ perl -pi'.orig' -e 's/bar/baz/' fileA     # backup to 'fileA.orig'
                $ perl -pi'*.orig' -e 's/bar/baz/' fileA    # backup to 'fileA.orig'
```

Quiz 10



From the shell, saying

```
    $ perl -p -i.orig -e "s/foo/bar/; ... "
```

is the same as using the program:

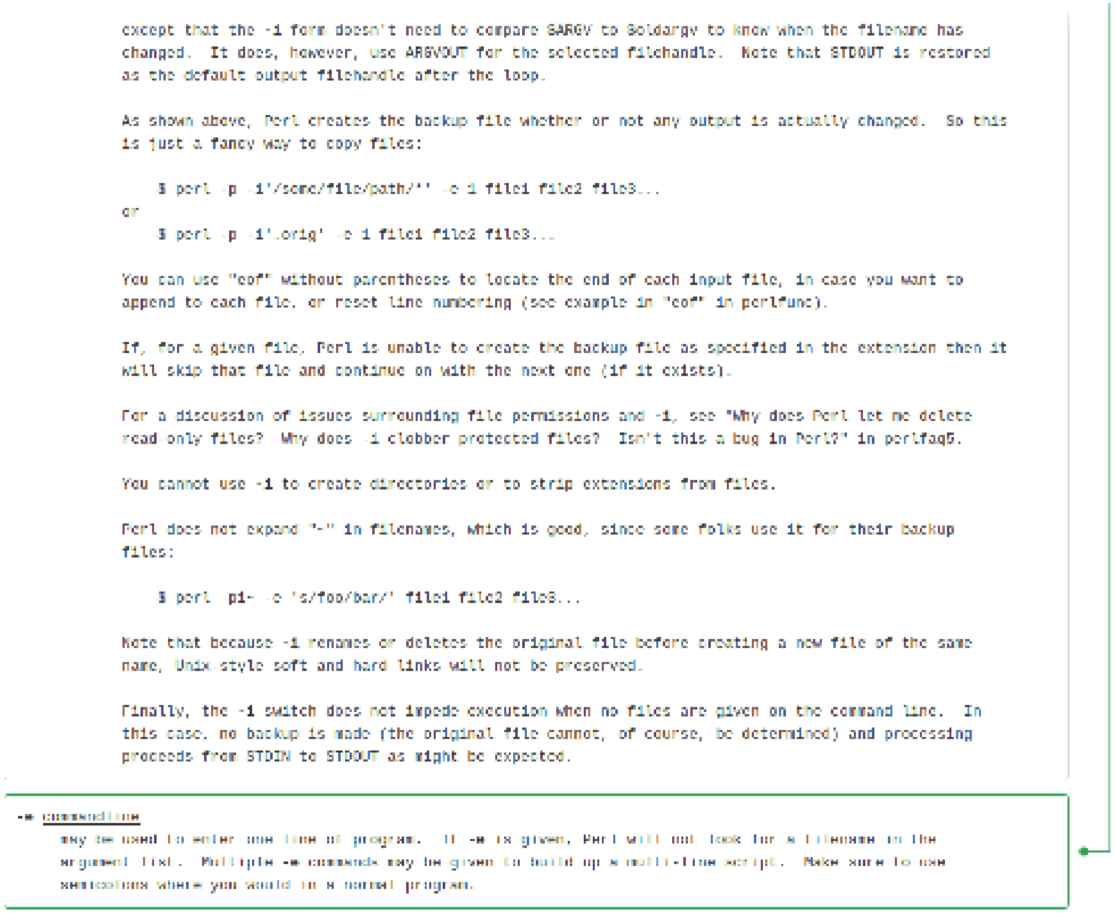
```
    #!/usr/bin/perl -pi.orig
    s/foo/bar/;
```

which is equivalent to

```
    #!/usr/bin/perl
    $extension = '.orig';
    LINE: while (<>) {
        if ($ARGV ne $oldargv) {
            if ($extension !~ /\*/) {
                $backup = $ARGV . $extension;
            }
            else {
                ($backup = $extension) =~ s/\*/$ARGV/g;
            }
            rename($ARGV, $backup);
            open(ARGVOUT, ">$ARGV");
            select(ARGVOUT);
            $oldargv = $ARGV;
        }
        s/foo/bar/;
    }
    continue {
        print;  # this prints to original filename
    }
    select(STDOUT);
```

except that the -i form doesn't need to compare $ARGV to $oldargv to know when the filename has

Quiz 10



except that the -i form doesn't need to compare $ARGV to $oldargv to know when the filename has changed. It does, however, use ARGVOUT for the selected filehandle. Note that STDOUT is restored as the default output filehandle after the loop.

As shown above, Perl creates the backup file whether or not any output is actually changed. So this is just a fancy way to copy files:

```
    $ perl -p -i'/some/file/path/*' -e 1 file1 file2 file3...
or
    $ perl -p -i'.orig' -e 1 file1 file2 file3...
```

You can use "eof" without parentheses to locate the end of each input file, in case you want to append to each file, or reset line numbering (see example in "eof" in perlfunc).

If, for a given file, Perl is unable to create the backup file as specified in the extension then it will skip that file and continue on with the next one (if it exists).

For a discussion of issues surrounding file permissions and -i, see "Why does Perl let me delete read-only files? Why does -i clobber protected files? Isn't this a bug in Perl?" in perlfaq5.

You cannot use -i to create directories or to strip extensions from files.

Perl does not expand "~" in filenames, which is good, since some folks use it for their backup files:

```
    $ perl -pi~ -e 's/foo/bar/' file1 file2 file3...
```

Note that because -i renames or deletes the original file before creating a new file of the same name, Unix-style soft and hard links will not be preserved.

Finally, the -i switch does not impede execution when no files are given on the command line. In this case, no backup is made (the original file cannot, of course, be determined) and processing proceeds from STDIN to STDOUT as might be expected.

-e commandline
    may be used to enter one line of program. If -e is given, Perl will not look for a filename in the argument list. Multiple -e commands may be given to build up a multi-line script. Make sure to use semicolons where you would in a normal program.

Perl like Grep, Sed and Awk functions

```
# check perl --help
# -e means single line expression (a raw regular expression is in fact
an executable expression in perl)
# -n means execute on each line
# -p means execute on each line and print the result
# -F... means split the source text using the following pattern ...
# -a is part of -F, and splits the source text into @F[...]
# -l means print everything with a separator, by default newlines
```

Perl like Grep, Sed and Awk functions

# check perl --help
# -e means single line expression (a raw regular expression is in fact an executable expression in perl)
# -n means execute on each line
# -p means execute on each line and print the result
# -F... means split the source text using the following pattern ...
# -a is part of -F, and splits the source text into @F[...]
# -l means print everything with a separator, by default newlines

Grep like:

```
perl -ne 'print if /chr1_geneA/' example2.fasta | more
perl -ne 'print if /chr1_geneB/' example2.fasta | more
```

Perl like Grep, Sed and Awk functions

# check perl --help
# -e means single line expression (a raw regular expression is in fact an executable expression in perl)
# -n means execute on each line
# -p means execute on each line and print the result
# -F... means split the source text using the following pattern ...
# -a is part of -F, and splits the source text into @F[...]
# -l means print everything with a separator, by default newlines

sed like:

```
perl -pe 's/chr1/chr2/' example2.fasta | more (without replacement)
perl -i -pe 's/chr1/chr2/' example2.fasta | more (with replacement)
```

Perl like Grep, Sed and Awk functions

# check perl --help
# -e means single line expression (a raw regular expression is in fact an executable expression in perl)
# -n means execute on each line
# -p means execute on each line and print the result
# -F... means split the source text using the following pattern ...
# -a is part of -F, and splits the source text into @F[...]
# -l means print everything with a separator, by default newlines

awk like:

```
cat /etc/passwd | awk -F: '{ print $1 }'
cat /etc/passwd | perl -F: -lane 'print @F[0]'
```

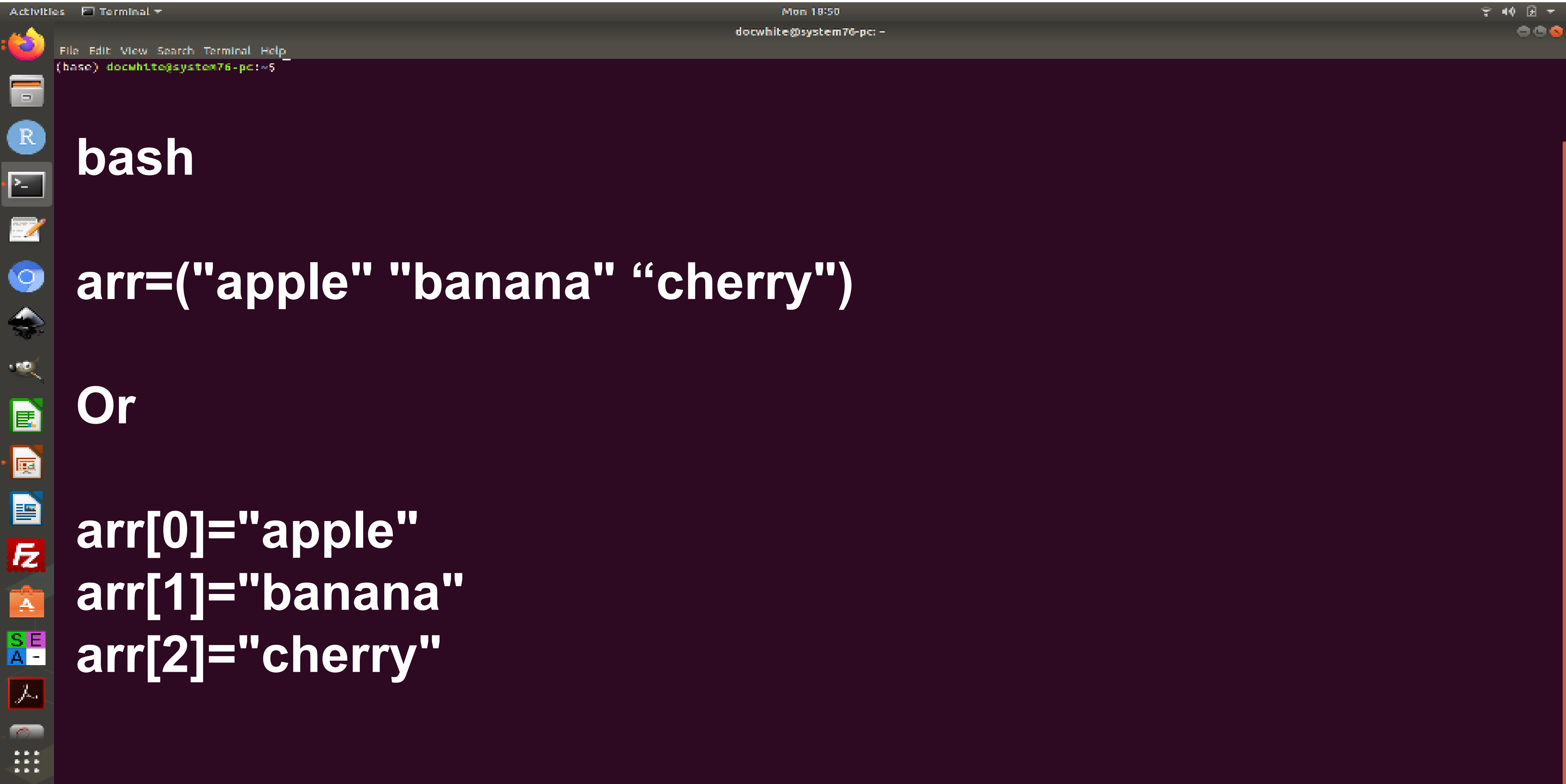
Array Variables in Bash

An array is a variable containing multiple values. Any variable may be used as an array.

There is no maximum limit to the size of an array, nor any requirement that member variables be indexed or assigned contiguously.

Arrays are zero-based: the first element is indexed with the number 0.

BASH arrays



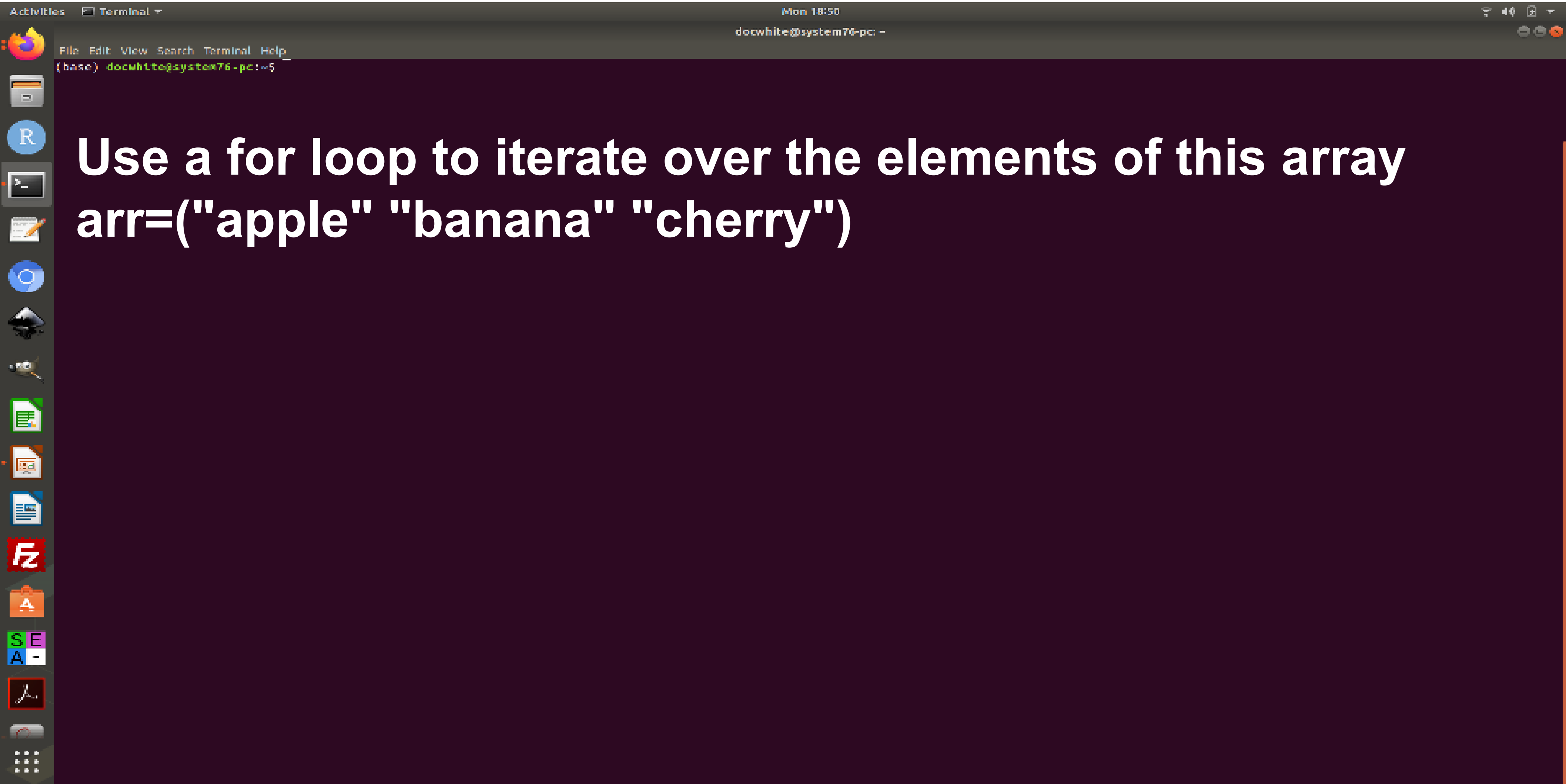
```
(base) docwhite@system76-pc:~$
```

```
bash

arr=("apple" "banana" “cherry")
```

Or

```
arr[0]="apple"
arr[1]="banana"
arr[2]="cherry"
```

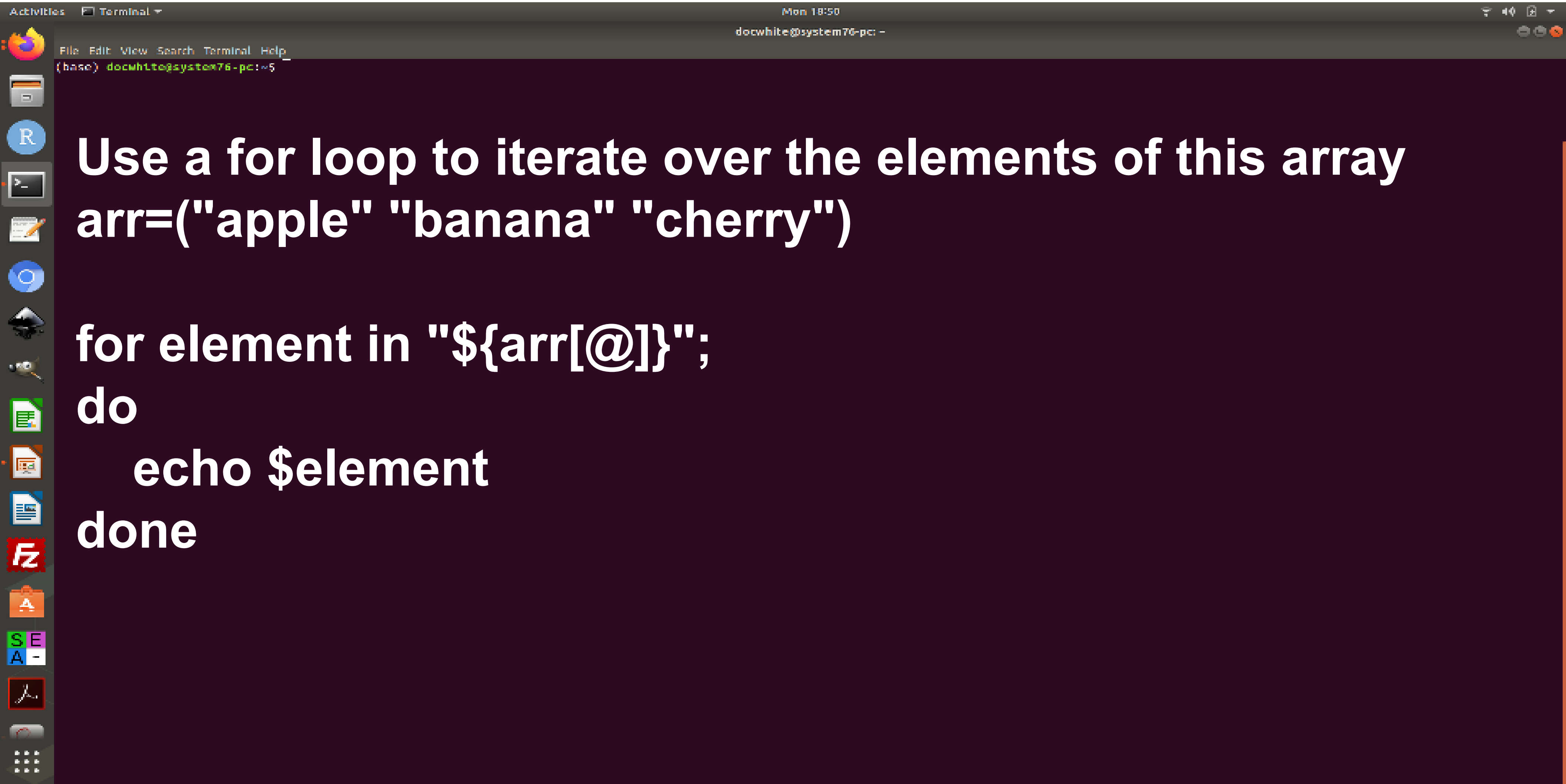
BASH for loop in arrays



```
(base) docwhite@system76-pc:~$
```

Use a for loop to iterate over the elements of this array
arr=("apple" "banana" "cherry")

BASH for loop in arrays

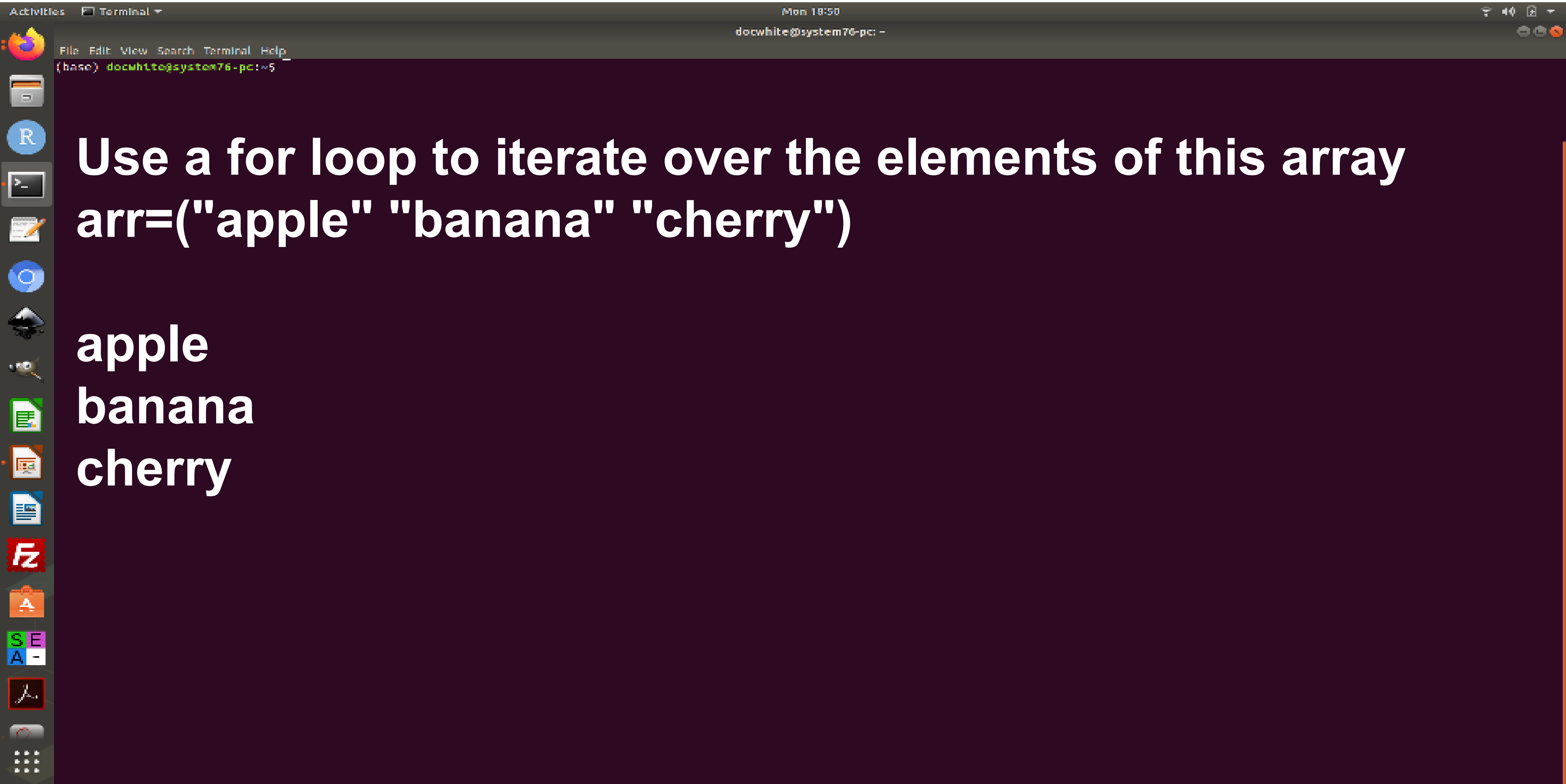


```
(base) docwhite@system76-pc:~$
```

Use a for loop to iterate over the elements of this array

```
arr=("apple" "banana" "cherry")

for element in "${arr[@]}";
do
    echo $element
done
```

BASH for loop in arrays

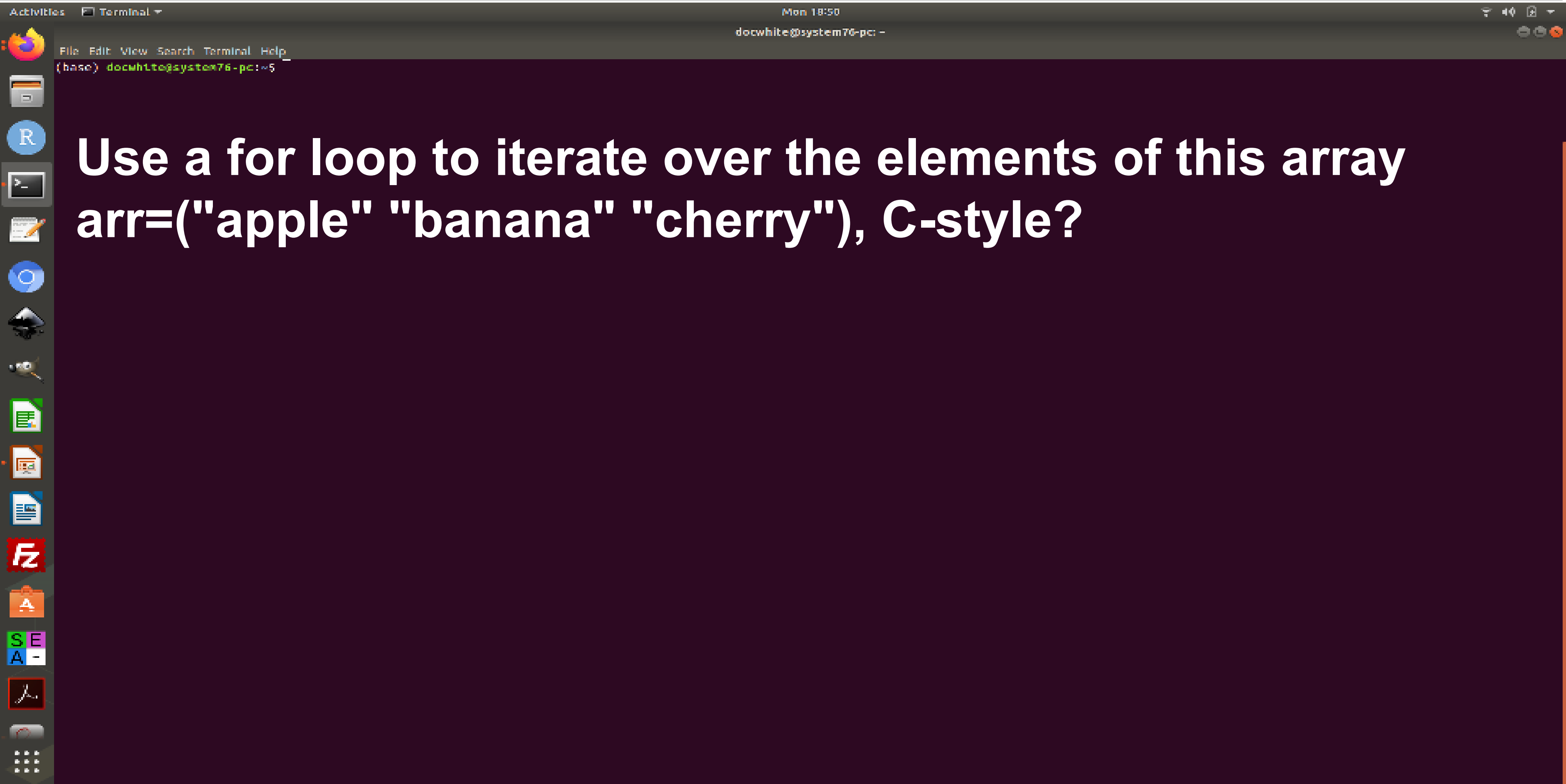


```
(base) docwhite@system76-pc:~$
```

Use a for loop to iterate over the elements of this array
arr=("apple" "banana" "cherry")

```
apple
banana
cherry
```

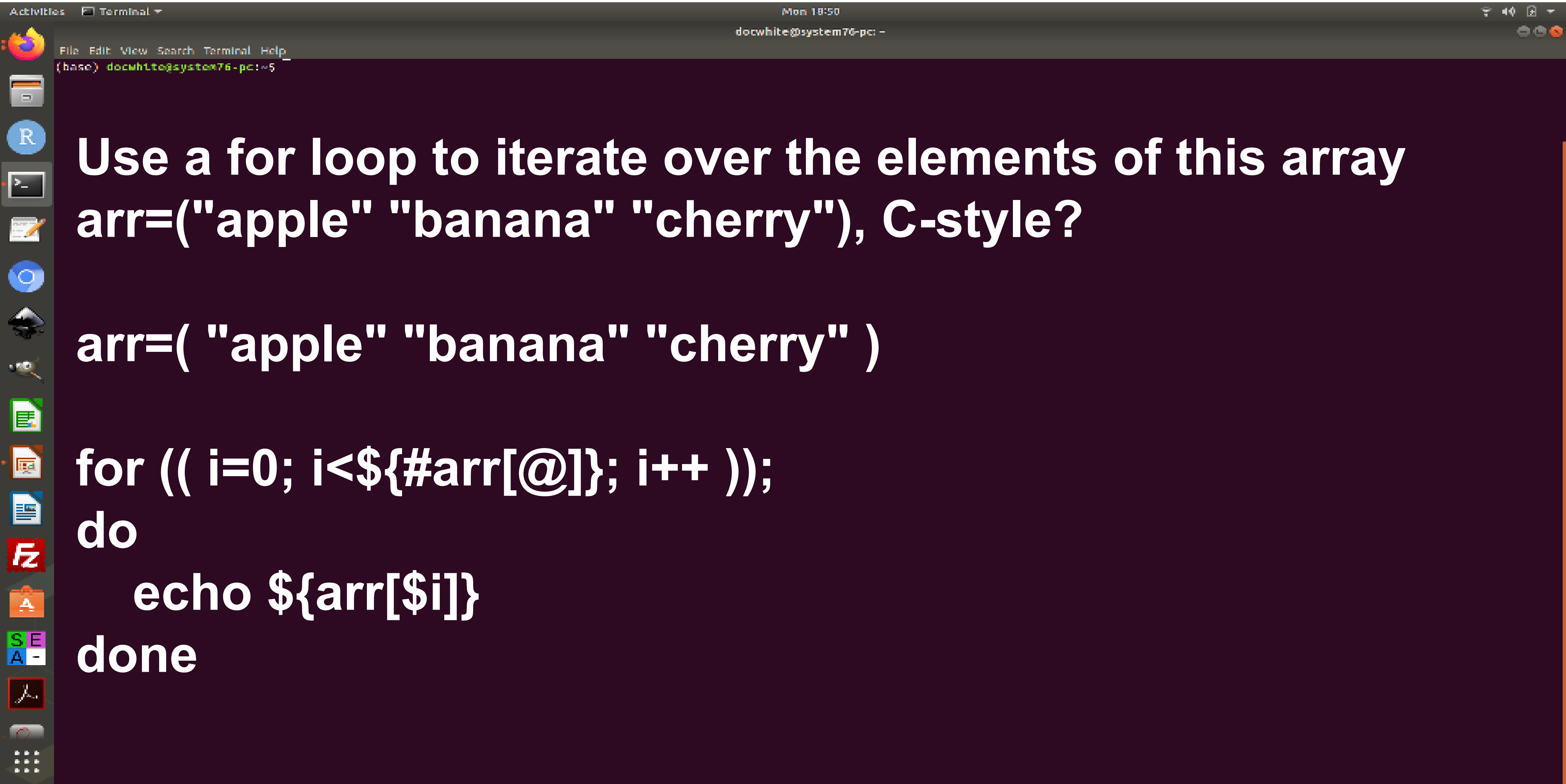
BASH for loop in arrays



```
(base) docwhite@system76-pc:~$
```

Use a for loop to iterate over the elements of this array arr=("apple" "banana" "cherry"), C-style?

BASH for loop in arrays

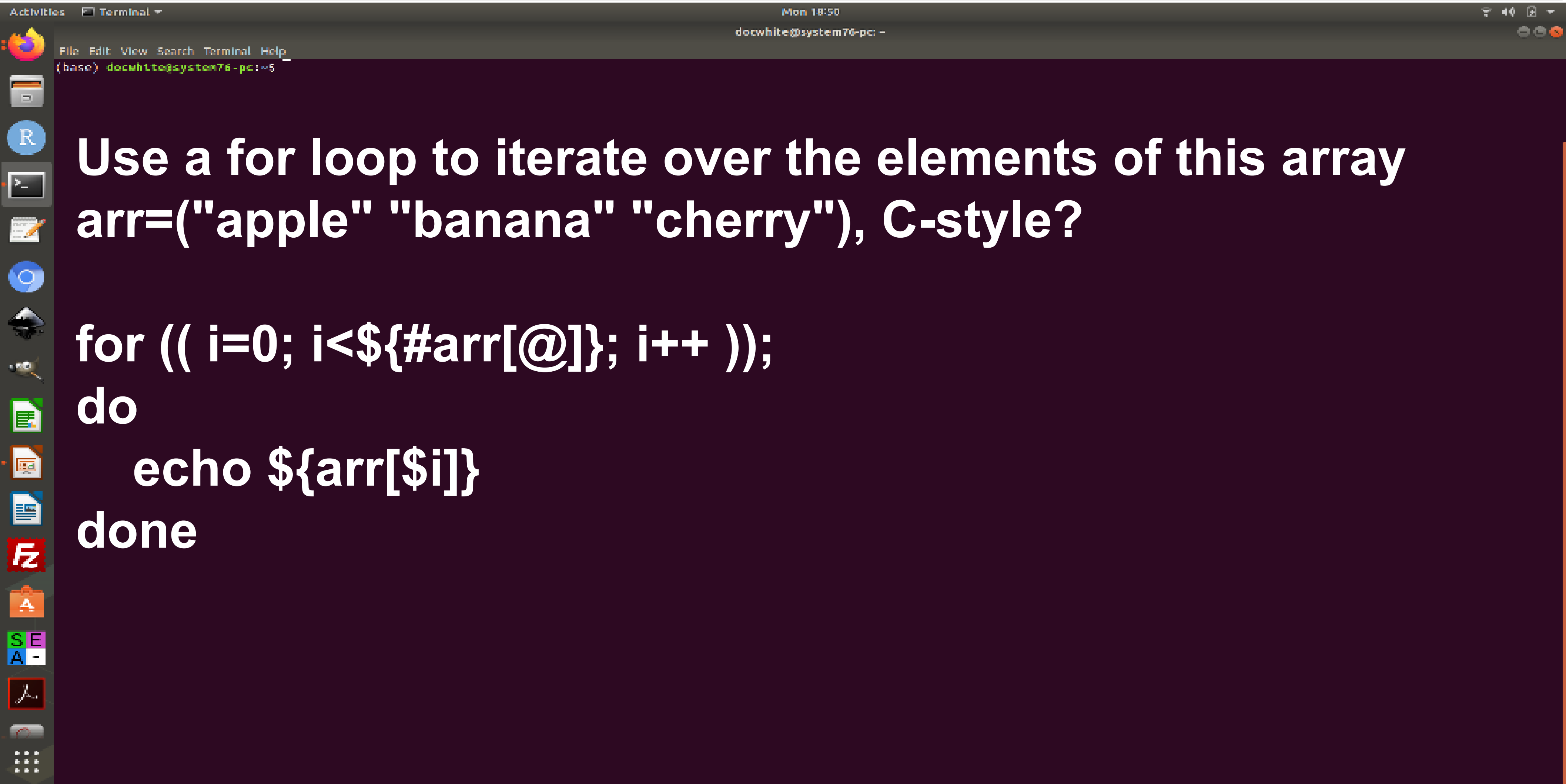


```
(base) docwhite@system76-pc:~$
```

Use a for loop to iterate over the elements of this array arr=("apple" "banana" "cherry"), C-style?

```
arr=( "apple" "banana" "cherry" )

for (( i=0; i<${#arr[@]}; i++ ));
do
    echo ${arr[$i]}
done
```

BASH for loop in arrays

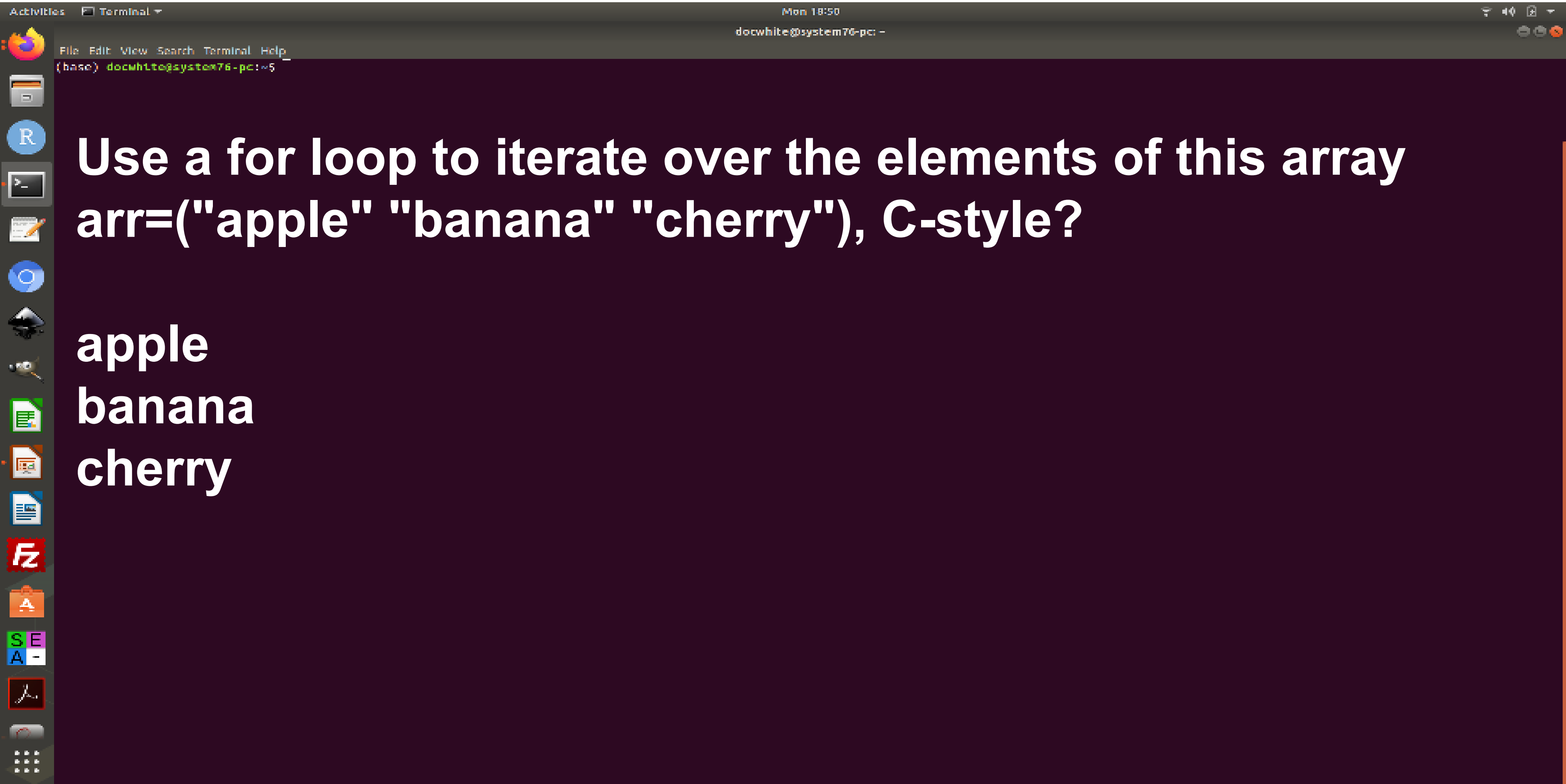


```
(base) docwhite@system76-pc:~$
```

Use a for loop to iterate over the elements of this array arr=("apple" "banana" "cherry"), C-style?

```
for (( i=0; i<${#arr[@]}; i++ ));
do
    echo ${arr[$i]}
done
```

BASH for loop in arrays

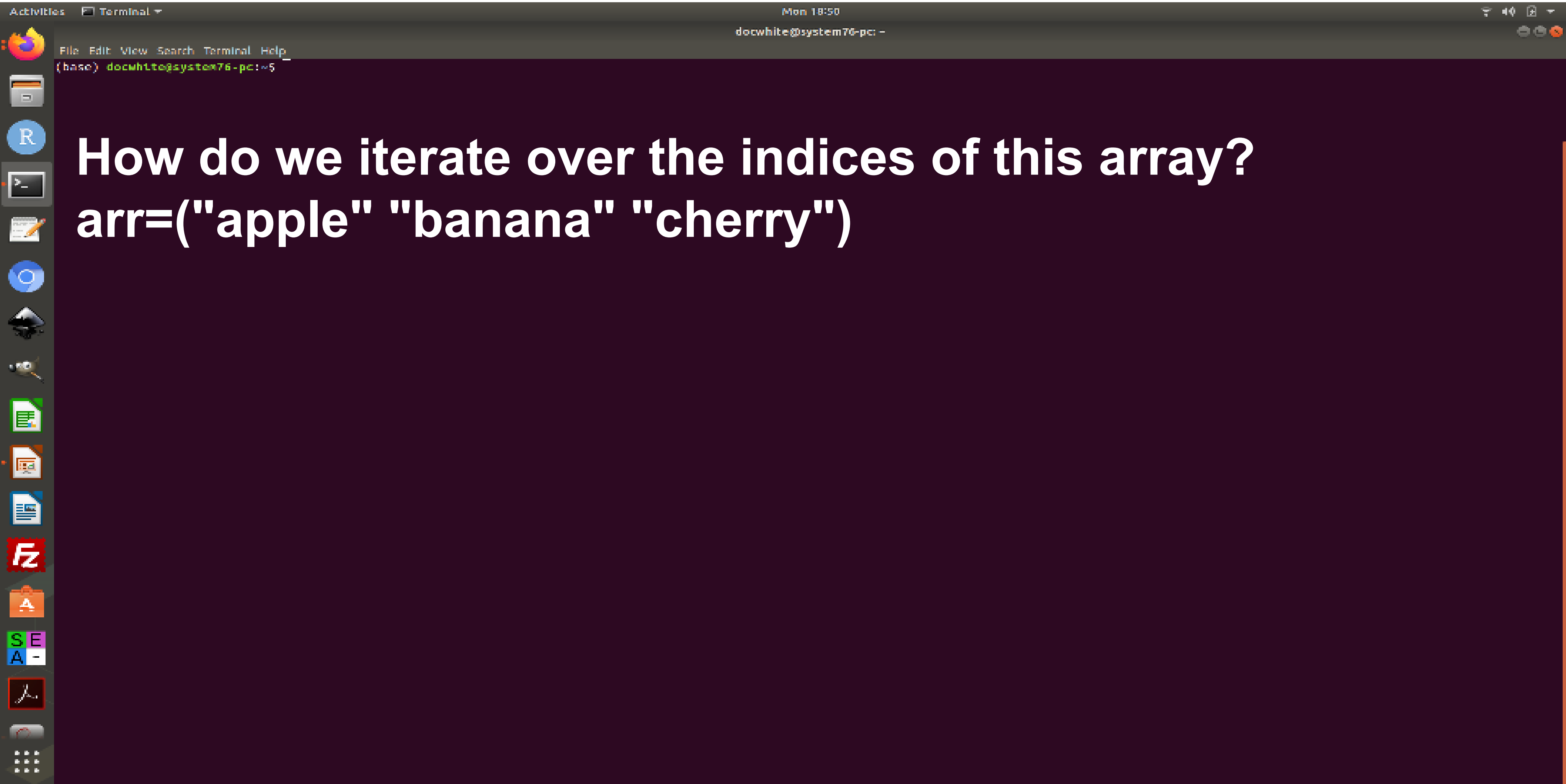


```
(base) docwhite@system76-pc:~$
```

Use a for loop to iterate over the elements of this array arr=("apple" "banana" "cherry"), C-style?

```
apple
banana
cherry
```

BASH for loop in arrays

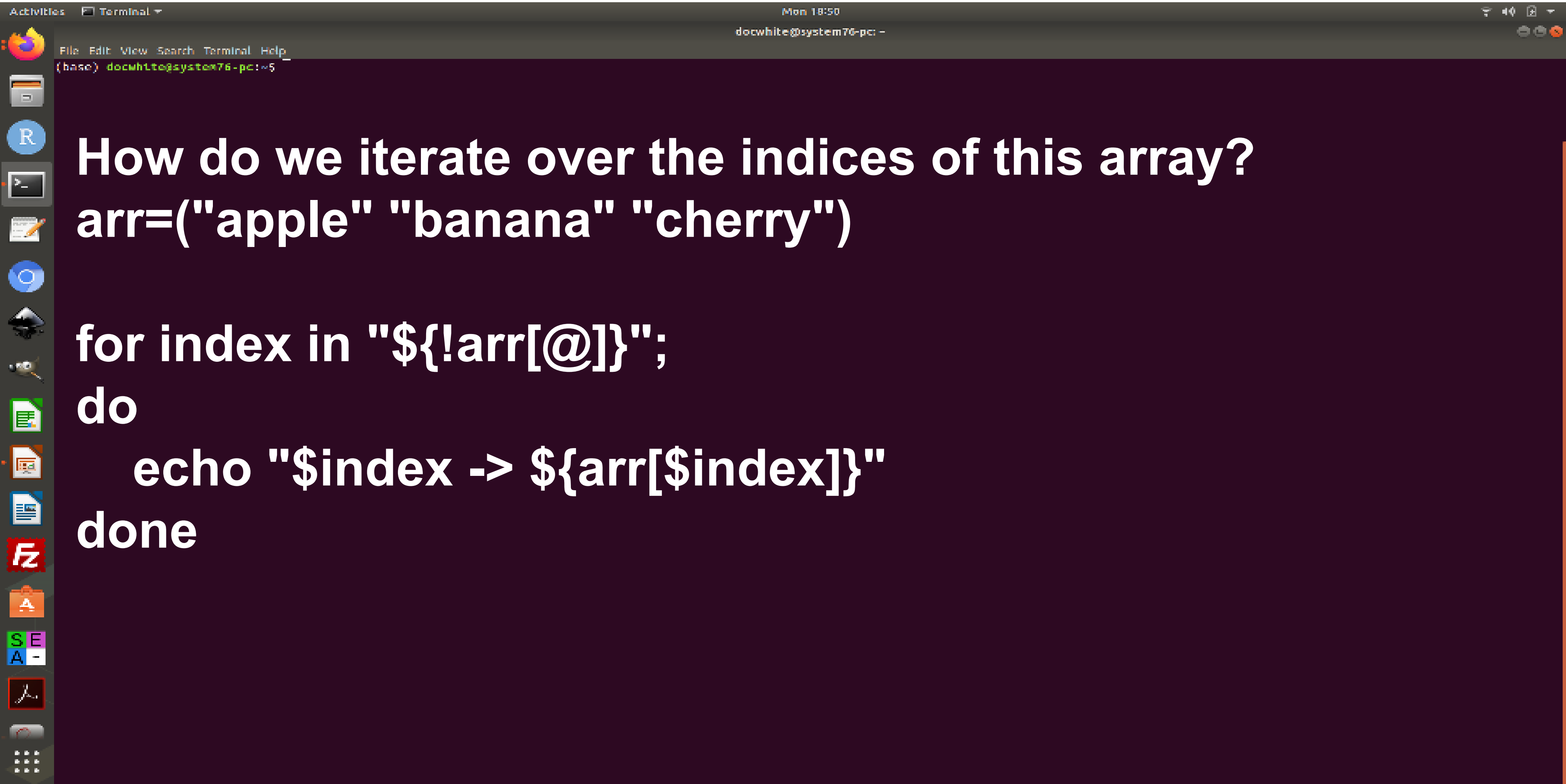


```
(base) docwhite@system76-pc:~$
```

How do we iterate over the indices of this array?

arr=("apple" "banana" "cherry")

BASH for loop in arrays

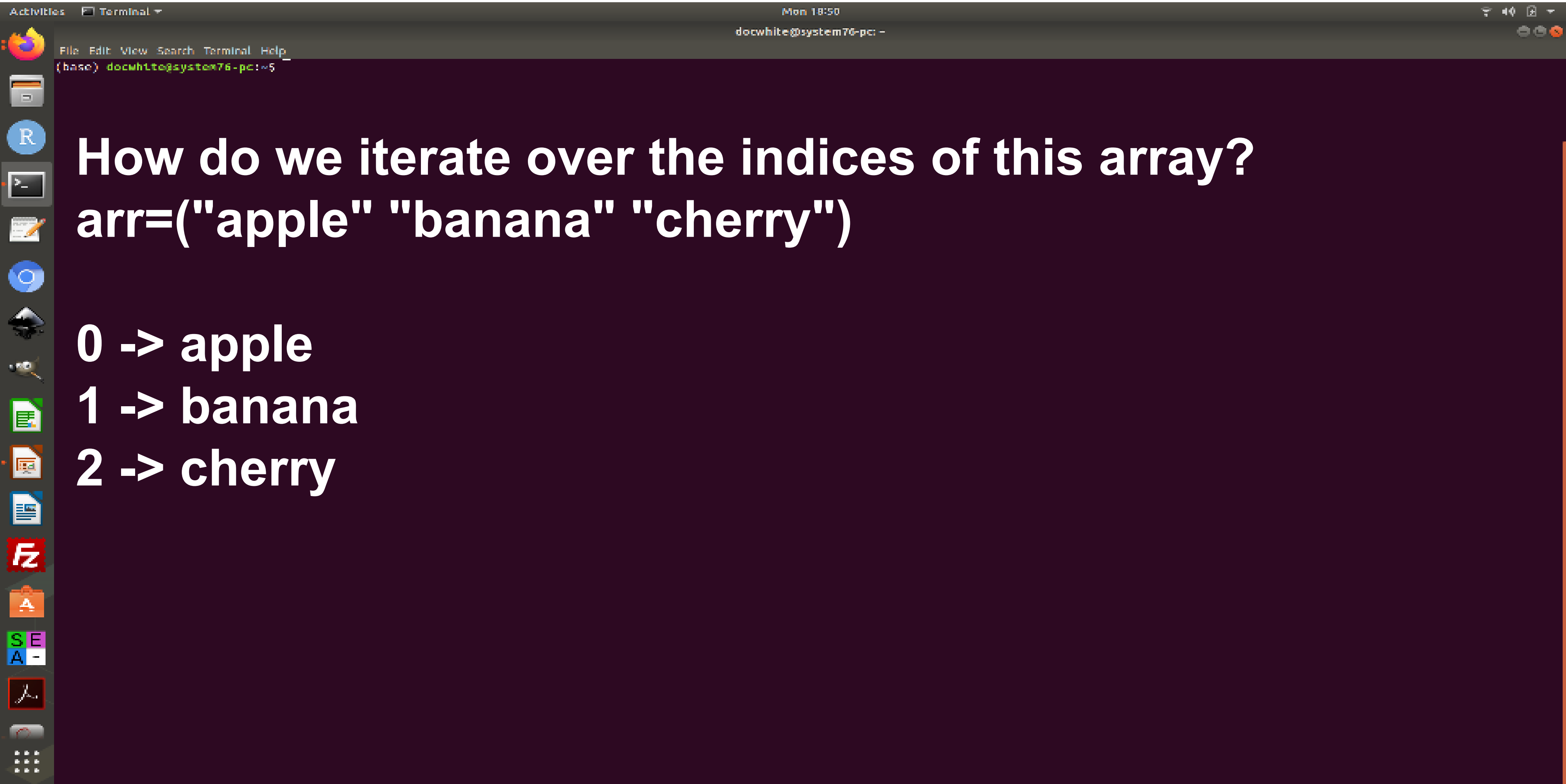


```
(base) docwhite@system76-pc:~$
```

How do we iterate over the indices of this array?

```
arr=("apple" "banana" "cherry")

for index in "${!arr[@]}";
do
    echo "$index -> ${arr[$index]}"
done
```

BASH for loop in arrays



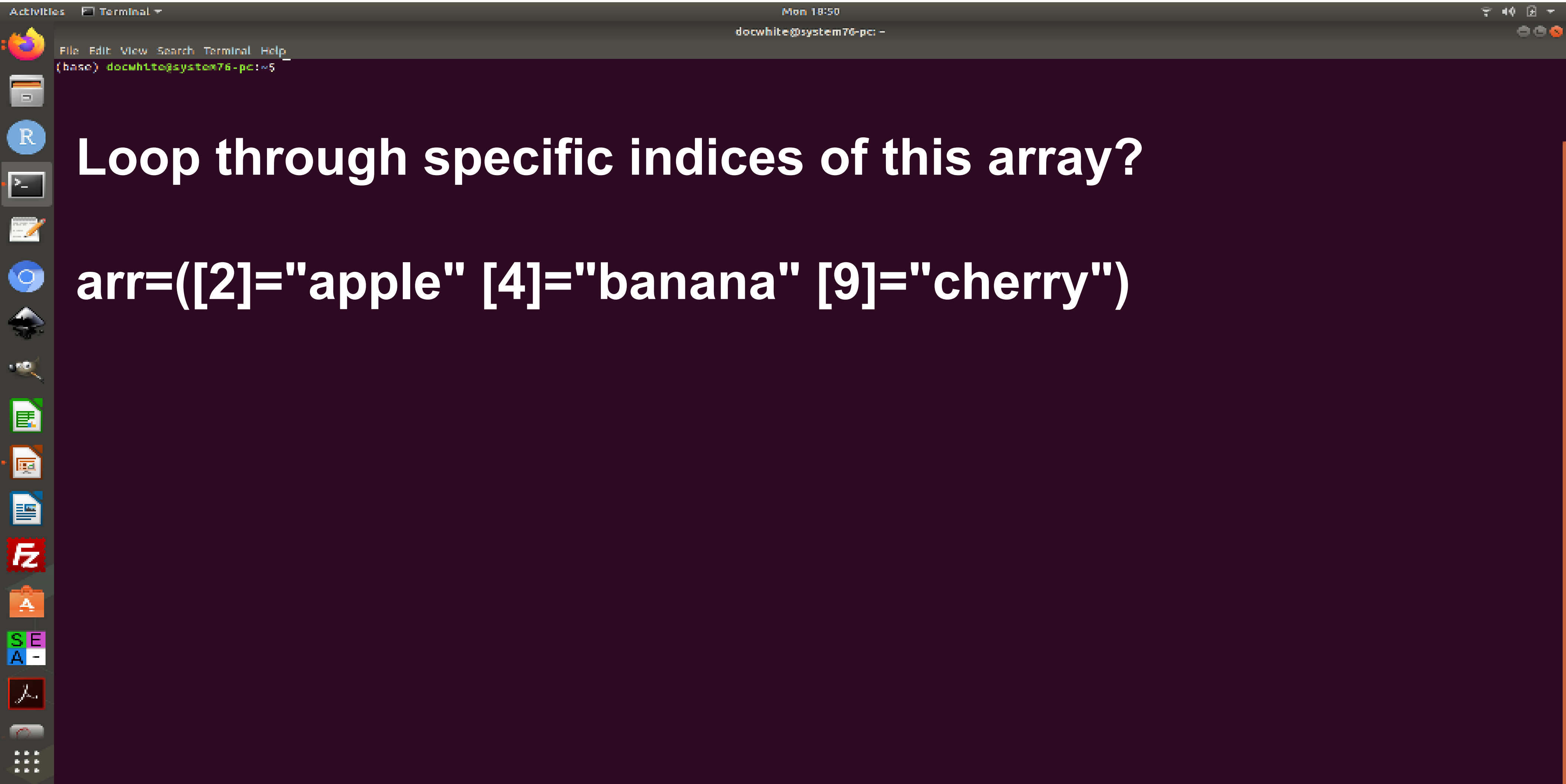
```
(base) docwhite@system76-pc:~$
```

How do we iterate over the indices of this array?

arr=("apple" "banana" "cherry")

0 -> apple
1 -> banana
2 -> cherry

BASH for loop in arrays

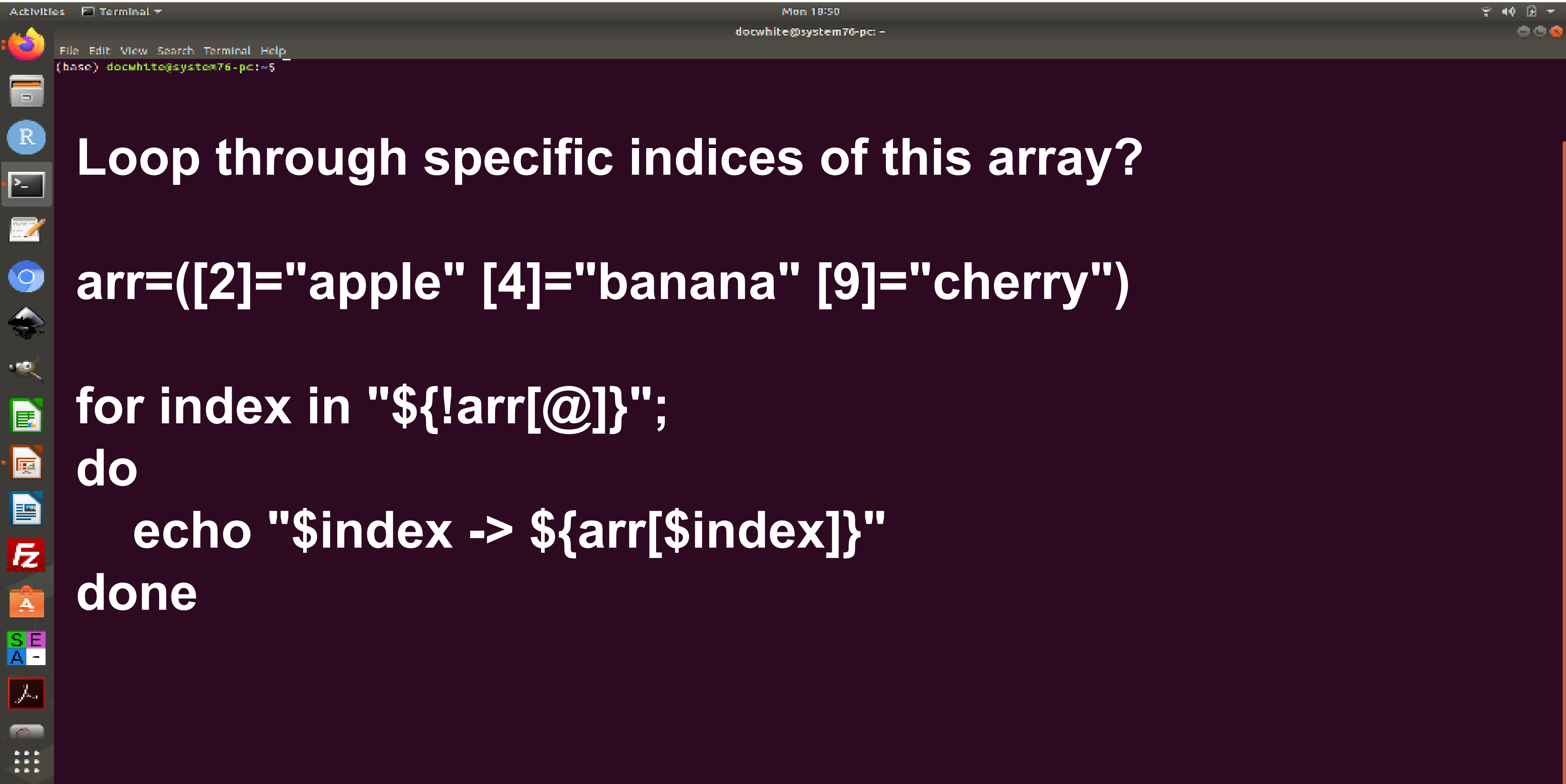


```
(base) docwhite@system76-pc:~$
```

Loop through specific indices of this array?

```
arr=([2]="apple" [4]="banana" [9]="cherry")
```

BASH for loop in arrays

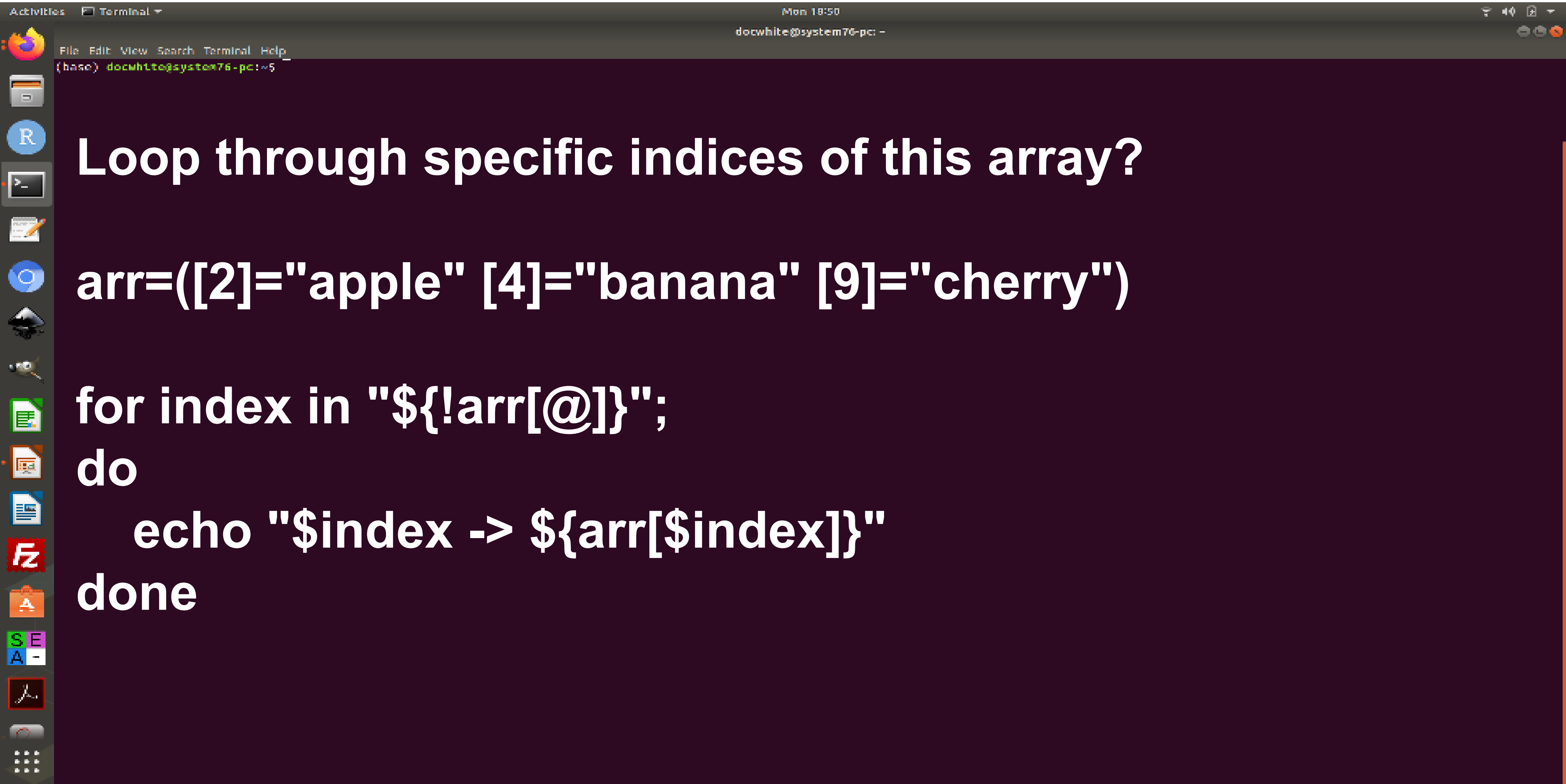


```
(base) docwhite@system76-pc:~$
```

Loop through specific indices of this array?

```
arr=([2]="apple" [4]="banana" [9]="cherry")

for index in "${!arr[@]}";
do
    echo "$index -> ${arr[$index]}"
done
```

BASH for loop in arrays

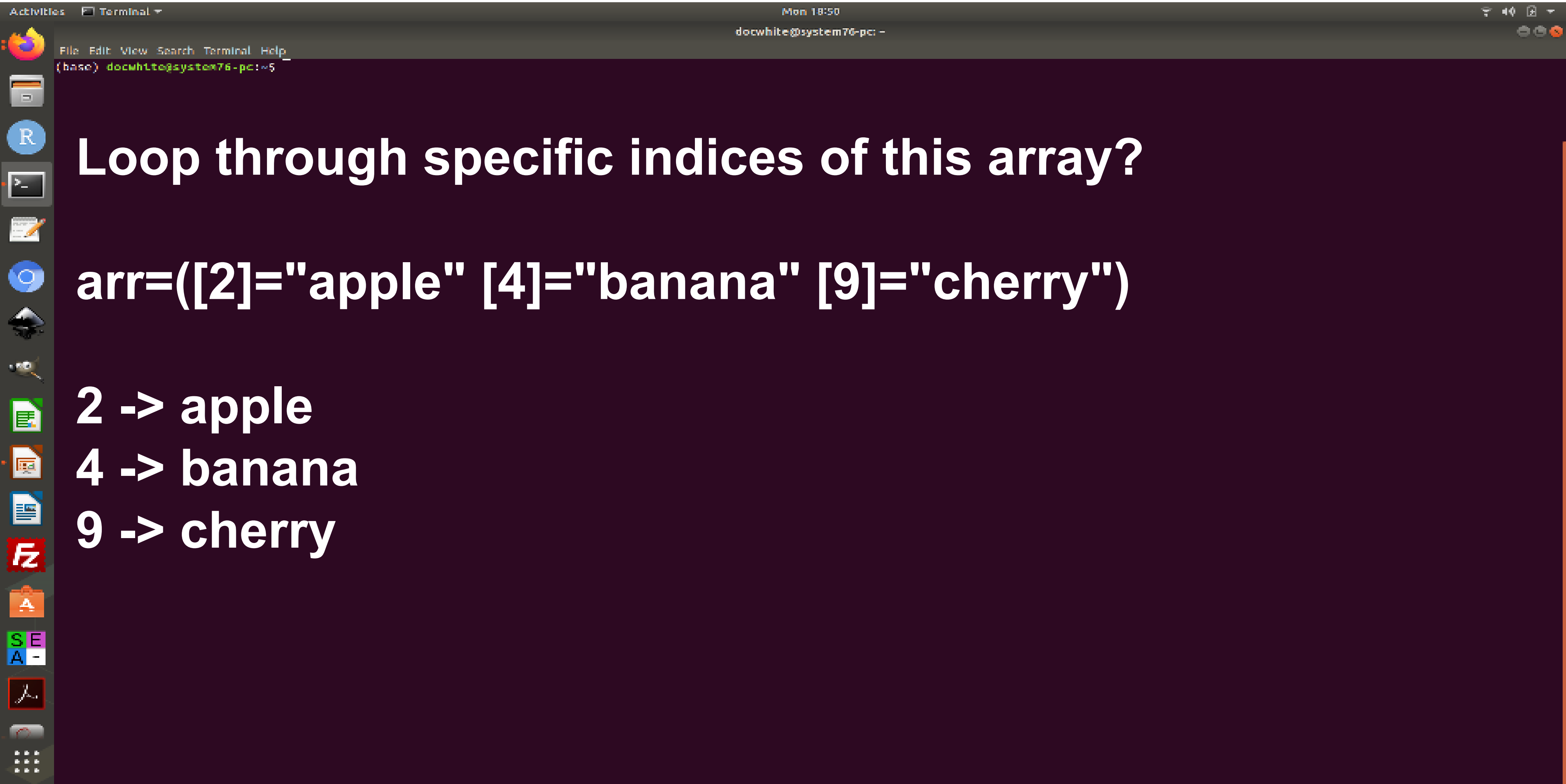


```
(base) docwhite@system76-pc:~$
```

Loop through specific indices of this array?

```
arr=([2]="apple" [4]="banana" [9]="cherry")

for index in "${!arr[@]}";
do
    echo "$index -> ${arr[$index]}"
done
```

BASH for loop in arrays



```
(base) docwhite@system76-pc:~$
```

Loop through specific indices of this array?

```
arr=([2]="apple" [4]="banana" [9]="cherry")
```

```
2 -> apple
4 -> banana
9 -> cherry
```

Question

Write a bash script to count the number of ATG (starts) and TAA, TAG, TGA (stops) from the example2.fasta file.

Remember that ATG encodes for methionine so the only count the from the beginning of the sequence or the end for the stops.

HOW WOULD YOU DO THIS?

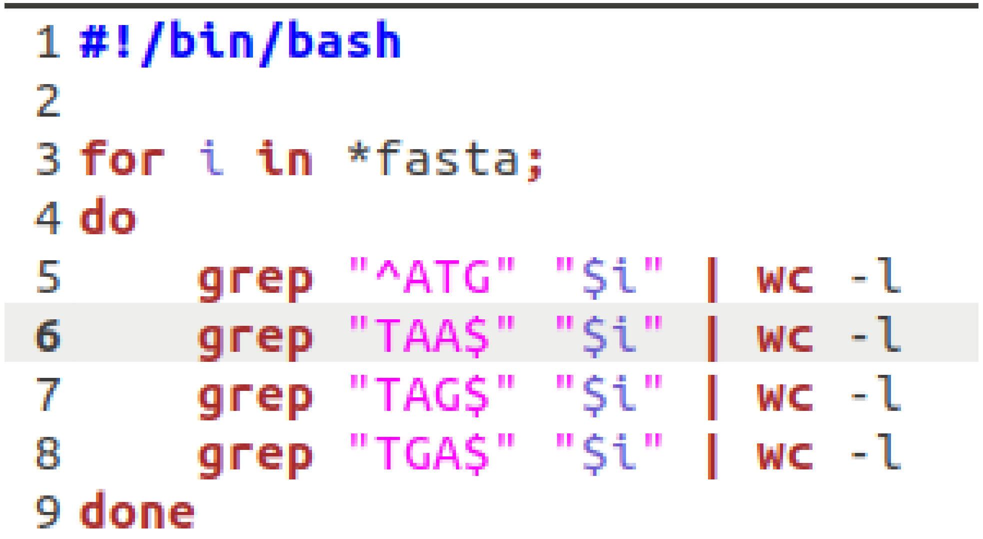
Question

Write a bash script to count the number of ATG (starts) and TAA, TAG, TGA (stops) from the example2.fasta file.

Remember that ATG encodes for methionine so the only count the from the beginning of the sequence or the end for the stops.

HOW WOULD YOU DO THIS?

BETTER WAY?



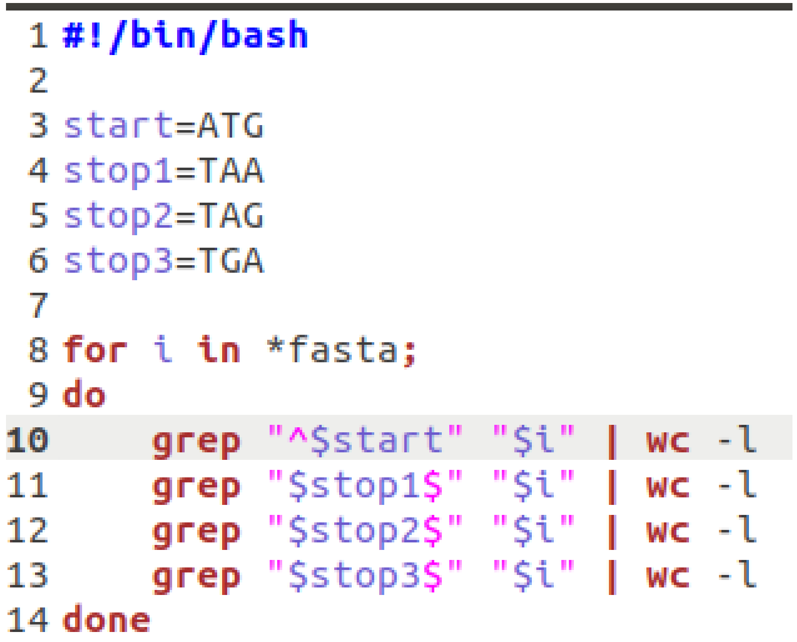
```bash
#!/bin/bash

for i in *fasta;
do
        grep "^ATG" "$i" | wc -l
        grep "TAA$" "$i" | wc -l
        grep "TAG$" "$i" | wc -l
        grep "TGA$" "$i" | wc -l
done
```

Question

Write a bash script to count the number of ATG (starts) and TAA, TAG, TGA (stops) from the example2.fasta file.

Remember that ATG encodes for methionine so the only count the from the beginning of the sequence or the end for the stops.

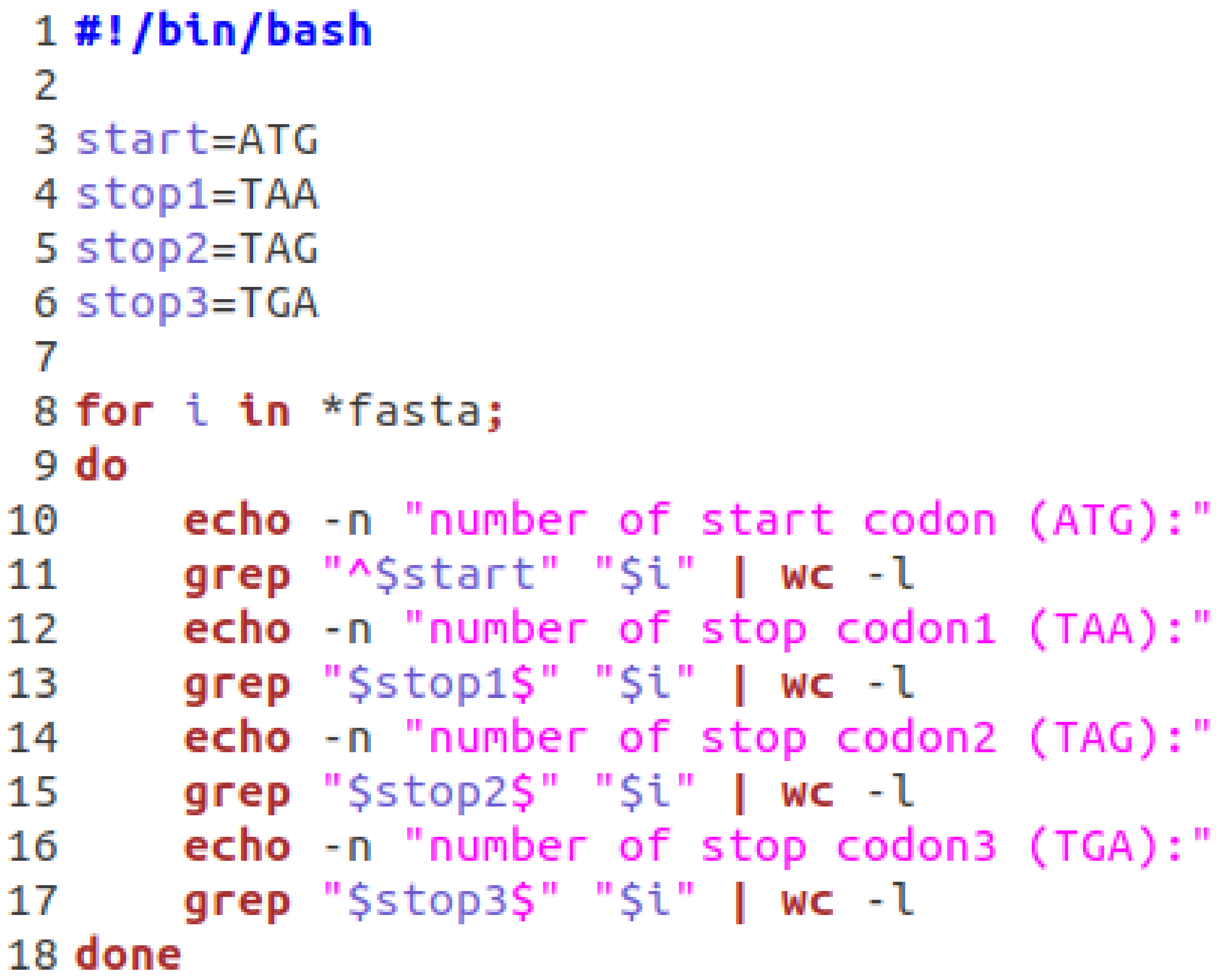
EVEN BETTER?



```bash
#!/bin/bash

start=ATG
stop1=TAA
stop2=TAG
stop3=TGA

for i in *fasta;
do
    grep "^$start" "$i" | wc -l
    grep "$stop1$" "$i" | wc -l
    grep "$stop2$" "$i" | wc -l
    grep "$stop3$" "$i" | wc -l
done
```

Question

EVEN BETTER?



```bash
#!/bin/bash

start=ATG
stop1=TAA
stop2=TAG
stop3=TGA

for i in *fasta;
do
	echo -n "number of start codon (ATG):"
	grep "^$start" "$i" | wc -l
	echo -n "number of stop codon1 (TAA):"
	grep "$stop1$" "$i" | wc -l
	echo -n "number of stop codon2 (TAG):"
	grep "$stop2$" "$i" | wc -l
	echo -n "number of stop codon3 (TGA):"
	grep "$stop3$" "$i" | wc -l
done
```

Question

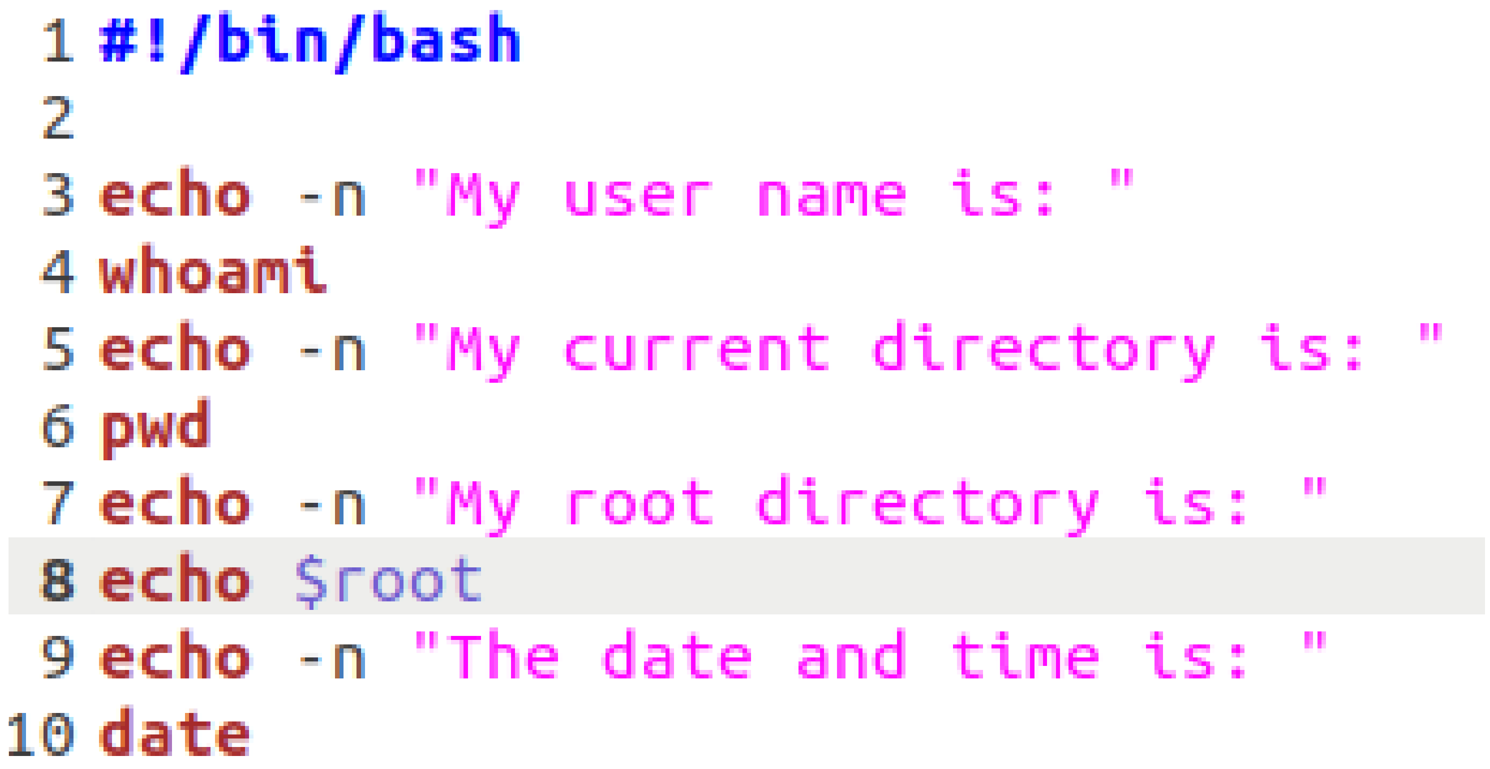
Write a bash script that tells me my username, current directory, the location of my root directory, and the date/time

HOW WOULD YOU DO THIS?

Question

Write a bash script that tells me my username, current directory, the location of my root directory, and the date/time

HOW WOULD YOU DO THIS?



```bash
#!/bin/bash

echo -n "My user name is: "
whoami
echo -n "My current directory is: "
pwd
echo -n "My root directory is: "
echo $root
echo -n "The date and time is: "
date
```

Question

Write a bash script that tells me my username, current directory, the location of my root directory, and the date/time

HOW WOULD YOU DO THIS?

```
bash script_date.sh
```

```
My user name is: docwhite
My current directory is: /home/docwhite/Desktop
My root directory is:
The date and time is: Tue Sep 28 19:40:29 EDT 2021
```

BASH - for loop

```
for i in file.*;do
  command $i
done
```

BASH - while loop

```
while[ condition ]
do
  command1
  command2
  command3
done
```

BASH - while loop

Command1 to Command3 will be executed repeatedly till condition is **false**. The argument for a while loop can be any boolean expression. Infinite loops occur when the conditional never evaluates to false. The while loop should be used as long as a certain condition is true, such as the a counter is less than a maximum value or the ping time to a server is lower than a threshold or forever if you loop while TRUE or while 1.

Here is the while loop one-liner syntax:

while [condition]; **do** commands; **done**

while control-command; **do** COMMANDS; **done**

BASH - while loop

```
#!/bin/bash
x=1
while [ $x -le 5 ]
do
  echo "Welcome $x
times"
  x=$(( $x + 1 ))
done
```

BASH - while loop

```
#!/bin/bash
x=1
while [ true ]
do
  echo "Welcome $x
times"
  x=$(( $x + 1 ))
done
```

BASH - while loop infinite

```
#!/bin/bash
while :
do
  echo "An Infinite loop”
done
```

BASH - while loop (one - liner)

```
x=1; while [ $x -le 5 ]; do echo "Welcome
$x times"; $(( x++ )); done
```

BASH - while loop (read line by line)

```
#!/bin/bash
FILE=$1
# read $FILE using the file descriptors
exec 3<&0
exec 0<$FILE
while read line
do
    # use $line variable to process line
    echo $line
done
exec 0<&3
```

BASH - while loop (in array)

```
#!/bin/bash
arr=( "apple" "banana" "cherry" )

i=0
len=${#arr[@]}
while [ $i -lt $len ];
do
    echo ${arr[$i]}
    let i++
done
```

BASH - until loop

The until loop is similar to the while loop but with reverse logic. Instead of looping while a condition is true you are assuming the condition is false and looping until it becomes **true.** They are reverse of each other in logical expression.

```
until [ CONDITION ]; do
  LINES OF CODE
  MORE LINES OF CODE
done
```

BASH - until loop

```
#!/bin/bash
NUM=1
until [ “$NUM” -gt 1000 ]; do
    echo $NUM
    let NUM=NUM*2
done
```

BASH - functions

```
Function_name( ){
    command
}
```

Think of a function as a small script within a script.
It's a small chunk of code which you may call
multiple times within your script.

MY FAVORITE WAY! (There is another way)

BASH - functions

```
Function function_name( ){
    command
}
```

Not my favorite. But, you may like it?

BASH – Passing Arguments/Return values

```
#!/bin/bash

print_this() {
  echo Hello $1
    return 5
}
print_this Mars
print_this Jupiter
echo The previous function has a return value of $?
```

BASH – Passing Arguments/Return values

Output

```
Hello Mars
Hello Jupiter
print_this Jupiter
The previous function has a return value of 5
```

Quiz 11

- On canvas now

Bonus 11

- Write a function that will return the number of lines it has in it?