1. [Bash Shell Scripts](https://www.tocode.co.il/bundles/bash-shell-scripts)

שתי הפקודות המרכזיות ב Bash לביצוע לולאות הן for ו while. בפרק זה נראה את שתי הפקודות, ההבדלים ביניהן ושימוש יעיל בכל אחת. בנוסף נלמד כיצד להפעיל פקודות אריתמטיקה בסיסיות כדי לשנות ערכי משתנים בתוך ומחוץ ללולאות

לולאות While.

לולאת while מזכירה מאוד פקודת if, רק הפעם מדובר בפקודת if שחוזרת על עצמה עד לכשלון. אם לדוגמא פקודת if מסוגלת לבדוק פעם אחת האם קובץ ריק ולקבל החלטה איך להתקדם, פקודת while תבדוק את אותו הדבר שוב ושוב ותמשיך לבצע את גוף הלולאה כל עוד התנאי מתקיים. תחביר בסיסי לבלוק while נראה כך:

# 1. perform command\_to\_check.

# 2. If $? == 0 perform the block

# 3. Otherwise continue to "done"

# 4. Repeat 1-3

while command\_to\_check

do

cmd1

cmd2

cmd3

done

הקוד הבא בודק האם משתמש בשם jimmy מחובר באמצעות הפקודה who, אך במקום לבדוק ולהמשיך הקוד ממשיך בלולאה עד שג׳ימי לא יופיע ברשימה:

# Check "jimmy" is not in the output of "who"

# NOTE: our grep command does not check position or whole words,

# so if a user named jimmyjimmy is connected it'll pass our test

while who | grep jimmy &> /dev/null

do

sleep 5

done

echo "Bye bye jimmy”

ניתן להשתמש בפקודת test (או בכתיב הסוגריים המרובעים) בתוך לולאת while, בדיוק באותו האופן בו שילבנו פקודה זו עם if. הנה דוגמא לתוכנית לא מאוד מאובטחת אך ממחישה את הנקודה:

read -p "What is the password ?" PASSWORD

while [[ $PASSWORD != "shhh" ]]; do

read -p "Wrong! Try again: " PASSWORD

done

echo "Welcome, Master”

לולאות while משמשות אותנו לביצוע פעולה כאשר איננו יודעים מראש את אורכה. לדוגמא המתנה עד שנוצר או נמחק קובץ מסוים (או מופעלת תוכנית מסוימת). מקרים נפוצים נוספים הינם פענוח של פרמטרים שהועברו בשורת הפקודה או קריאת קובץ או קלט אחר שורה אחר שורה. את הקוד לשתי הדוגמאות נראה בפרקים הבאים.

לולאות For:

לולאת for משלבת בין שני דברים: בלוק של קוד ואלמנט מתוך רשימה. הלולאה תרוץ על רשימה של אלמנטים ותפעיל את הקוד פעם אחת על כל אלמנט ברשימה. תחביר הפקודה נראה כך:

for NAME in WORDS

do

# Now the variable $NAME is the iteration variable

cmd1

cmd2

cmd3

done

לדוגמא הלולאה הבאה מדפיסה שורת טקסט עבור כל אחד מהפירות ברשימה:

FRUITS="Apple Banana Orange Strawberry"

for ITEM in ${FRUITS}

do

echo I want to eat an ${ITEM}

done

כפי שאתם יכולים לראות, הרשימה היא בסוך הכל שורת טקסט שכל האיברים מופרדים ברווחים. רשימה כזו בדיוק אנו יכולים לקבל מסימני ה Wildcards של Bash. כך, אפשר להשתמש בלולאת for כדי לעבור על קבצים. הקוד הבא לדוגמא, ישנה את הסיומת של כל קובץ txt בתיקייה לסיומת: bak.

for TXTFILE in \*.txt

do

mv "${TXTFILE}" "${TXTFILE%txt}bak"

done

אלו מכם שמגיעים משפות תכנות אולי מצפים לראות ב-for אפשרות להגדיר מונה רץ, ואכן יש ל Bash את האפשרות הזו על ידי אופרטור הסוגריים העגולים הכפולים. כך זה נראה בקוד:

for (( i=0; i < 10; i+=2 )); do

echo "Counting: $i"

done

אריתמטיקה:

שימוש נוסף בסימן הסוגריים הכפולים הינו לצורך חישובים אריתמטיים. כל ביטוי שתכתבו בתוך סימן זה יפוענח על ידי Bash כביטוי אריתמטי, באופן שמזכיר שפות תכנות מודרניות. בתוך הסוגריים גם אין צורך להשתמש ב-$ כדי לקבל ערך של משתנה.  
בדוגמא הראשונה אנו מבצעים חישוב לתוך משתנה, ללא שימוש בסימן ה-$, והערך זמין לפקודות Shell בהמשך התוכנית:

$ (( x = 5 ))

$ (( x += 27 ))

$ (( x \*= 2 ))

$ echo $x

64

בדוגמא השניה אנו מונים את מספר הקבצים בהם יש יותר מ-10 שורות:

#!/usr/bin/env bash

for FILENAME in \*.txt

do

LINES=$(wc -l $FILENAME| tr -s ' ' | cut -d ' ' -f 2)

if [[ $LINES -gt 10 ]]; then

(( counter += 1 ))

fi

done

echo "Found $counter files"

תנאים:

מטרתה של ה Shell הינה להפעיל תוכניות אחרות, ולכן פקודות תנאי ב Shell אינה בודקת האם תנאי מסוים השתערך לאמת או שקר, אלא האם תוכנית מסוימת הצליחה בפעולתה או נכשלה. בפרק זה נלמד כיצד לעבוד עם תנאים ב Shell ובמה זה שונה מכתיבת פקודות תנאי בשפות תכנות אחרות.

מטרתה של ה Shell הינה להפעיל תוכניות אחרות, ולכן פקודות תנאי ב Shell אינה בודקת האם תנאי מסוים השתערך לאמת או שקר, אלא האם תוכנית מסוימת הצליחה בפעולתה או נכשלה.  
כן, קראתם טוב: כל פקודת תנאי ב Shell מחייבת הפעלה של תוכנית אחרת. בלוק ה then יופעל אם התוכנית האחרת הצליחה, ובלוק ה-else יופעל אם התוכנית נכשלה.  
התנהגות זו מחייבת אותנו לתשובה על השאלה איך יודעים אם תוכנית הצליחה או נכשלה. קחו לדוגמא את התוכנית ls. די ברור שהפעלת הפקודה:

ls /etc

חשבת להצלחה. עושה רושם גם כי הפעלת הפקודה על קובץ לא קיים, למשל:

ls /foo/bar

נחשבת לכשלון. אבל מה לגבי הפעלה מסוג זה:

ls /etc /foo/bar

הפקודה גם הדפיסה את כל תוכן התיקייה /etc וגם הציגה הודעת שגיאה. האם היתה כאן הצלחה או כשלון? ומי בכלל קובע מה נקרא הצלחה או כשלון של תוכנית?  
  
התשובה היא שבעולם היוניקס כל תוכנית צריכה להחזיר לדווח בסיום הפעלתה האם היא הצליחה או נכשלה. אם נסתכל על תוכנית לדוגמא בשפת C, נוכל לראות כי כל תוכנית צריכה לכלול פונקציית main המחזירה ערך מספרי:

int main(int argc, char \*argv[])

{

return 0;

}

ערך ההחזר של התוכנית הוא שקובע האם היא הצליחה או נכשלה. החזרת המספר 0 נחשבת להצלחה, ואילו החזרת כל ערך שאינו 0 נחשבת לכשלון. ערך ההחזר של כל תוכנית שאנו מפעילים נשמר בתוך המשתנה המיוחד $? (שימו לב שמאחר ומדובר במשתנה אחד, ערכו נדרס בכל פעם שמפעילים תוכנית).  
נתבונן שוב בדוגמאות שהצגנו עם התוכנית ls. שימו לב שאני מפנה את כל הפלט (שגיאות וגם פלט תקין) לקובץ המיוחד /dev/null.  הסיבה היא שלא מעניין אותי מה התוכנית מדפיסה, או האם ואיזה הודעות שגיאה היא מייצרת, אלא אך ורק ערך ההחזר שלה.

$ ls /etc &>/dev/null; echo $?

0

$ ls /foo/bar &>/dev/null; echo $?

1

$ ls /etc /foo/bar &>/dev/null; echo $?

1

עכשיו זה ברור: ההפעלה הראשונה הצליחה ואילו שתי האחרות נכשלו. זכרו: מי שמחליט האם תוכנית הצליחה או לא זהו כותב התוכנית, והוא מדווח על כך באמצעות החזרת ערך הצלחה או כשלון.

הפקודה if:

תחביר הפקודה if ב bash נראה כך:

if <command>

then

<success block>

else

<error block>

fi

זכרו שניתן להשתמש בפקודה help כדי לקבל הסבר קצר על הפקודה יחד עם התחביר המלא. כך למשל עבור הפקודה if:

help if

הפקודה if מקבלת פקודה אחרת, מפעילה את התוכנית שקיבלה ובודקת: אם התוכנית הצליחה (כלומר ערך ההחזר שווה 0) יתבצע בלוק ה then, ואם התוכנית נכשלה יתבצע בלוק ה else. בלוק ה else הוא אופציונאלי. הבה נמחיש באמצעות דוגמא

# Example 1: Check if a file exists (not the best way to do this)

#

# check if the command "ls /etc/passwd" succeeds

if ls /etc/passwd &>/dev/null

then

# when an ls command succeeds, it means all files it tried to display were available

echo “File /etc/passwd is in place”

else

# when the ls command fails, it means the file/folder is not there

echo “Error locating file: /etc/passwd”

fi

# Example 2: Find a regexp inside a file

#

# since grep succeeds if the regexp was found in the file

# we can use it to search in files

if ! grep bash /etc/passwd &>/dev/null

then

echo “Nobody uses bash…”

fi

הפקודה test:

גישת הפעלת התוכניות לא נראית רעה ממבט ראשון, אך מהר מאוד אנו צפויים להתקל בבעיה: ראינו כבר שהמשתנה $# כולל את מספר הפרמטרים שהועברו לסקריפט. מה יקרה אם נרצה לבדוק שאכן הועברו לסקריפט שני פרמטרים או יותר. Shell לא מספקת תחביר לבדיקת תנאים.  
במקרה כזה אנו נעזר בפקודת Shell מיוחדת שנקראת test (וניתן להפעילה גם באמצעות סוגריים מרובעים כפולים). פקודה זו מחזירה ערך הצלחה אם התנאי שקיבלה השתערך כאמיתי, או ערך כשלון אם התנאי שקיבלה השתערך כשקרי.  
נתבונן בדוגמא הבאה:

$ FOO=5; BAR=7; BUZ=5

$ test $FOO -eq 5; echo $?

0

$ test $BAR -eq 10; echo $?

1

$ test $FOO -lt $BAR; echo $?

0

גם על הפקודה test אפשר למצוא מידע באמצעות הפקודה help test. קיצור דרך נוח לפקודה זו הוא פקודת הסוגריים המרובעים (או בגירסת Bash, הסוגריים המרובעים הכפולים). את אותה הדוגמא שראינו ניתן לכתוב גם באופן הבא:

$ FOO=5; BAR=7; BUZ=5

$ [[ $FOO -eq 5 ]]; echo $?

0

$ [[ $BAR -eq 10 ]]; echo $?

1

$ [[ $FOO -lt $BAR ]]; echo $?

0

מילות התנאי -eq, -gt, -lt בודקות שמספר מסוים שווה, קטן מ או גדול מ מספר אחר. משלימות אותן מילות התנאי >, < ו= הפועלות על מחרוזות. בנוסף לבדיקות על מחרוזות ומספרים, הפקודה test כוללת גם אוסף רחב של בדיקות על קבצים, לדוגמא:

# check if /etc/passwd is a file

$ [[ -f /etc/passwd ]]

# check if /tmp is a directory

$ [[ -d /tmp ]]

# check /tmp/test is a non-empty file

$ [[ -s /tmp/test ]]

# check file /etc/group is readable

$ [[ -r /etc/group ]]

הפקודה case:

מילת התנאי האחרונה לפרק זה היא המילה case המשמשת אותנו לבדיקת משתנה אל מול מספר תבניות. כך זה נראה:

case $1 in

\*.png) echo "It's a png image";;

\*.txt) cat $1;;

\*.mp3) echo "Cool music";;

esac

הכתיב קצת מוזר. בתרגום לעברית הקוד שכתבנו אומר:

1. התווים שמשמאל לסוגר העגול מייצגים תבנית, ולכן אנו בודקים את הסיומת של המשתנה $1
2. לכל סיומת מבצעים פקודה אחרת, זו הפקודה שרשומה אחרי סימן הסוגריים. לדוגמא עבור קובץ mp3 נדפיס את הטקסט Cool music.
3. אנו מסיימים פקודה באמצעות סימן נקודה-פסיק שמופיע פעמיים.

## קריאה באמצעות לולאת while

בפרק זה נלמד כיצד להשתמש בלולאת while בשילוב פקודת read כדי לכתוב פילטרים, כלומר תוכניות המקבלות קלט מ STDIN, מדפיסות פלט ל STDOUT וניתן להשתמש בהן בצד ימין של סימן ה-|. לאחר מכן נראה כיצד לבצע הפניות מתוך הסקריפט עצמו, כך נוכל להפעיל תוכנית חיצונית מתוך הסקריפט, לפענח את הפלט שלה ולפי התוצאה להחליט כיצד להמשיך. זו גם תהיה הדוגמא המסכמת של הפרק.

מימוש פילטר באמצעות לולאת while:

הפקודה read קוראת שורת קלט מ stdin ומכניסה אותה לתוך משתנה. הפקודה נכשלת אם אין יותר קלט זמין (כלומר משתמש הקליד תו-סוף-קלט או שה pipe נסגר). מכאן שקל לשלב פקודה זו עם פקודת while לכתיבת תוכניות פשוטות, למשל הצגת קובץ או ספירת שורות. הנה שתי דוגמאות להתחלה:

#!/usr/bin/env bash

# A program that prints each input line

# (similar to cat)

# Note the quotes around $LINE when printing - without them you'll get

# surprising results when the line has special shell characters

while read LINE

do

echo "$LINE"

done

# A program that counts all input lines

# (similar to wc -l)

while read LINE

do

(( count++ ))

done

echo "Input had $count lines"

הפעלה של תוכנית כזו יכולה להתבצע באופן רגיל משורת הפקודה, או באמצעות כתיבתה מימין לסימן ה-|. אם לדוגמא שמרתם את התוכנית השניה בשם mywc.sh תוכלו לכתוב את הפקודה הבאה לספירת השורות בקובץ /etc/shells:

$ cat /etc/shells | ./mywc.sh

קריאת תוכן קובץ מתוך הסקריפט:

במקרים רבים אנו לא מעוניינים שהמשתמש שמפעיל את הסקריפט יצטרך לבצע את ההפניה, ונרצה לשלב את הפונקציונליות של קריאת קובץ או פלט תוכנית חיצונית מתוך הסקריפט. התחביר ב bash הוא פשוט: הוסיפו את ההפניה בסיום לולאת ה while (אחרי הפקודה done) וההפניה תשפיע על כל הלולאה.

הבה נבחן 4 דוגמאות להפניות שונות אותן ניתן לרשום לאחר הפקודה done. נתחיל עם הפנית קלט מתוך קובץ, ונכתוב סקריפט שיציג את רשימת השורות שאינן הערות ואינן ריקות בקובץ /etc/shells. במידה ואחת השורות בקובץ מייצגת shell שאינו מותקן, הסקריפט יציג הודעת אזהרה. הנה קוד התוכנית:

#!/usr/bin/env bash

RES=0

while read SHELL

do

# skip empty lines

[[ -z $SHELL ]] && continue

# skip comment lines

[[ $SHELL == \#\* ]] && continue

if ! [[ -x $SHELL && -f $SHELL ]]

then

echo "Something is wrong with shell: $SHELL"

RES=1

fi

done < /etc/shells

exit $RES

נשים לב למספר נקודות לגבי התוכנית:

1. התוכנית מסתיימת בכשלון או הצלחה באמצעות העברת פרמטר לפקודה exit בסוף הסקריפט (שורה 21). סקריפטים כאלה קל לשלב בתוך סקריפטים אחרים במידת הצורך, ולבדוק האם הם הצליחו. ערך 0 מציין הצלחה וכל ערך אחר מציין כשלון.
2. בעת קריאת הקובץ לא כל השורות מעניינות אותנו. טכניקה מומלצת לדילוג על שורות לא רלוונטיות היא שימוש בפקודה continue, שממשיכה לאיטרציה הבאה של הלולאה. בדוגמא אנו מדלגים על שורות ריקות או על שורות המתחילות בסולמית.
3. לאחר הפקודה done השתמשנו הפניית stdin כדי לשלוח את תוכן הקובץ בתור מקור המידע לפקודה read.

הפניית פלט הלולאה לקובץ חיצוני:

אפשרות נוספת שניתן לכתוב לאחר הפקודה done היא חץ הפניית ה stdout, מה שיאפשר לנו להפנות את כל פלט הפקודה לקובץ חיצוני. התוכנית הבאה מקבלת שם קובץ ויוצרת קובץ חדש המכיל את כל השורות מהקובץ המקורי עם מספרי שורות בתחילתן:

#!/usr/bin/env bash

set noclobber

if [[ $# != 1 ]]

then

echo "Usage: $0 <filename>"

exit 1

fi

while read LINE

do

printf "%0.3d %s\n" $(( ++count )) "$LINE"

done < $1 > $1.numbered

נשים לב למספר נקודות לגבי הדוגמא:

1. מאחר ואנו מצפים שהמשתמש יזין שם קובץ, עלינו לוודא שכך אכן קרה. משתמש ששכח להקליד שם קובץ מצפה לקבל הודעת ״הוראות שימוש״ נעימה ולא התנהגות מפתיעה של הסקריפט.
2. הפקודה set noclobber בשורה 3 מונעת מ bash לדרוס קובץ קיים. כשמשתמשים בהפניית stdout לקובץ, הוספת שורה זו בראש הסקריפט היא דרך טובה לוודא שלא דרסנו בטעות קובץ חשוב (שאולי היה שם מבעוד מועד).

הפניית פלט הלולאה לתוכנית חיצונית:

ההפניה השלישית אותה ארצה להדגים כאן היא הפניית פלט כל הלולאה לתוכנית חיצונית. לצורך כך אנו נכתוב תוכנית המדפיסה על המסך את השורה הארוכה ביותר בקובץ ונשתמש בפקודה sort. הפקודה sort קוראת שורות מ stdin וממיינת אותן, ולכן אנו יכולים לשלוח אליה את כל האורכים ולקבל רק את השורה עם האורך הגדול ביותר. אם ההפניה היתה מתבצעת בכל שורת הדפסה, כל פקודת sort היתה מקבלת שורה אחת בלבד לעבוד עליה. מאחר ואנו שולחים את פלט הלולאה כולה, הפקודה sort יכולה למיין את כל השורות וכך לתת לנו את השורה הארוכה ביותר בלבד. הנה הקוד:

#!/usr/bin/env bash

while read LINE

do

echo ${#LINE}:"$LINE"

done | sort -rn | cut -d: -f2- | head -1

נשים לב למספר נקודות בקוד התוכנית:

1. אופן הפעולה של הסקריפט: הוספת שדה אורך לפני כל שורה בטקסט, שליחת השורות (יחד עם שדה האורך שלהן) לפקודה sort, חיתוך של שדה האורך באמצעות הפקודה cut ובסוף הדפסת שורת הפלט הראשונה מ sort באמצעות הפקודה head. יש כאן שילוב של הרבה תוכניות שיחד מאפשרות קוד יחסית קצר (אם כי לא הכי יעיל לפתרון הבעייה).
2. הפניית הפלט מלולאת ה while היא הבסיס להצלחת התוכנית. אם ה-| היה נכתב בשורת ה echo במקום, כל הדפסה היתה נשלחת לפקודת sort נפרדת, ואף פקודת sort לא היתה יכולה לראות את כל השורות ולמיין אותן.

 מעבר שורה-שורה על פלט פקודה חיצונית:

אפשרות רביעית ואחרונה הינה קריאה שורה-שורה של פלט פקודה חיצונית. ננסה לדוגמא לספור את הגודל הכולל של כל הקבצים בפלט התוכנית ls. נתחיל בנסיון ראשון שלא יציג את התוצאה הנכונה, אבל יראה לנו טעות נפוצה בשימוש בהפניות:

#!/usr/bin/env bash

# Calculate total size of all the files

# printed by ls

#

# WRONG SOLUTION - DOESN'T WORK

#

ls -l | while read \_ \_ \_ \_ SZ \_ \_ \_ FNAME

do

[[ -f $FNAME ]] || continue

(( total += $SZ ))

echo "Add $SZ (for file $FNAME). Total = $total"

done

echo "Total = $total"

פלט התוכנית בתיקייה אצלי נראה כך:

Add 288 (for file a.sh). Total = 288

Add 179 (for file shells). Total = 467

Add 0 (for file shells.numbered). Total = 467

Total =

ננסה להבין מה קרה כאן:

1. תחילה שימו לב למה שכן עובד: הפקודה READ יכולה להפריד את שורת הקלט למילים. מאחר וחלק מהמילים לא מעניינות אותי בתוכנית בחרתי להכניס את הערכים שלהן למשתנה \_. כך ברור יותר איזה מילים כן רלוונטיות, במקרה שלנו SZ ו FNAME.
2. בתוך גוף הלולאה ההדפסות נכונות - כלומר גם ערך המשתנה total גדל עם כל שורת קלט, וגם המשתנים SZ ו FNAME קיבלו ערכים נכונים.
3. ביציאה מהלולאה לעומת זאת איבדנו את הספירה, וכעת ערך המשתנה total הינו ריק.

הפעלת הלולאה מימין לסימן ה-| גרמה להפעלתה ב Sub Shell. המשתנים שהוגדרו שם, וגם שינוי תיקיית עבודה, אליאסים וכל פעולה אחרת שספציפית ל shell לא משפיעים על ה shell החיצוני. למרות השם הזהה, המשתנה total של שורות 13 ו 15 אינו אותו המשתנה כמו זה שאנו מנסים להדפיס בשורה 18. במקרה הכללי, כל הפעלה של תוכנית מימין לסימן ה-| גורמת להפעלתה בתוך sub shell.

כיצד ניתן לתקן את התוכנית? עלינו לקרוא את המידע מהתוכנית ls כפי שקראנו אותו מקבצים בדוגמאות הקודמות. התחביר ב bash להפעלת תוכנית חיצונית וקריאת המידע ממנה כאילו היתה קובץ מורכב מחץ וסוגריים. להלן גירסא מתוקנת לתוכנית שראינו:

#!/usr/bin/env bash

# Calculate total size of all the files

# printed by ls

#

while read \_ \_ \_ \_ SZ \_ \_ \_ FNAME

do

[[ -f $FNAME ]] || continue

(( total += $SZ ))

echo "Add $SZ (for file $FNAME). Total = $total"

done < <(ls -l)

echo "Total = $total"

הפעם במקום להפעיל את הלולאה משמאל לסימן ה-|, לולאת הקריאה מופעלת מאותו ה shell בו אנו נמצאים, והקלט שלה מגיע מתוך הפעלת התוכנית ls -l באמצעות החלפה שנקראת Process Substitution: פלט התוכנית מועבר כאילו נקרא מתוך קובץ לתוך הלולאה.

## חלוקת הקוד למספר קבצים

בואו נכתוב סקריפט המקבל שני פרמטרים, כל אחד מהם הינו שם קובץ, ומחליף את תכני הקבצים. נוודא גם שאכן קיבלנו שני פרמטרים אחרת נכתוב הודעת שגיאה:

#!/usr/bin/env bash

if [[ $# -ne 2 ]]

then

echo "Error: Expecting 2 arguments, got: $#"

exit 1

fi

mv $2 /tmp/$$

mv $1 $2

mv /tmp/$$ $1

במבט זריז בסקריפט ניתן לראות שמחציתו מוקדש לקוד יחסית גנרי, לוודא את מספר הארגומנטים שקיבל. כתיבת סקריפטים נוספים תביא אותנו למצב שאנו משכפלים את קוד הבדיקה הזה כמעט לכל סקריפט שנכתוב.

במצב כזה אנו נצא נשכרים אם נוכל להעתיק את הקוד הגנרי לפונקציה, ולהשתמש בה מכל אחד מהסקריפטים שלנו.

טעינת קובץ חיצוני:

בואו נעתיק את הקוד לפונקציה בקובץ חיצוני, נקרא לקובץ utils.sh. כך הוא נראה אצלי:

#############################

# check\_arguments

#

# checks the number of arguments got vs. number of

# arguments expected, and if there's a mismatch

# print an error message and exit

#

# Usage: check\_arguments <expected> <got>

#

function check\_arguments {

EXPECTED=$1

ACTUAL=$2

if [[ $EXPECTED -ne $ACTUAL ]]

then

echo "Error: Expecting $EXPECTED arguments. Got: $ACTUAL"

exit 1

fi

}

הקוד עצמו כמעט זהה לבדיקה שכבר כתבנו, מלבד מגבלה אחת: לא ניתן להשתמש ישירות במשתנים המיוחדים, מאחר ו-$# בתוך פונקציה כולל את מספר הפרמטרים שהועברו לפונקציה (ולא לתוכנית). מסיבה זו אנו מעבירים לפונקציה שני פרמטרים: לכמה פרמטרים התוכנית מצפה וכמה קיבלה בפועל. הפוקנציה כבר כוללת את קוד ההשוואה.

הקובץ utils.sh איננו קובץ הפעלה, לא צריך לתת לו הרשאות הפעלה וגם לא לרשום את שורת ה #! בראשו. אנו הולכים להשתמש בו בתור קובץ ספרייה ולטעון אותו תמיד מסקריפטים אחרים.

כעת אפשר לשנות את קובץ הסקריפט המקורי שלנו, במקום לבצע את כל הבדיקה הקובץ יטען את utils.sh ויפעיל את הפונקציה ממנו. אנו טוענים קובץ חיצוני באמצעות סימן הנקודה, וכך נראה קוד התוכנית לאחר השינוי:

#!/usr/bin/env bash

. utils.sh

check\_arguments 2 $#

mv $2 /tmp/$$

mv $1 $2

mv /tmp/$$ $1

חלוקת קוד למספר קבצים מצמצמת חזרה על קוד, ובכך מקלה על תחזוקת הסקריפטים שלנו.

הגדרת פונקציהfunction hello\_world { echo Hello World }

פונקציות מהוות דרך טובה לשימוש מחדש בקוד בכל שפת תכנות, ובפרט גם ב Bash. אנו מגדירים פונקציה על ידי כתיבת המילה function ולאחר מכן שם הפונקציה, בלי לציין את סוג ומספר הפרמטרים שהפונקציה מצפה לקבל. כך זה נראה:

function hello\_world {

echo Hello World

}

הפעלת הפונקציה מבוצעת באמצעות כתיבת שמה לאחר הגדרתה, למשל בהמשך הסקריפט ניתן להפעיל את הפונקציה שהגדרנו באמצעות:

hello\_world

טיפול בפרמטרים:

מאחר ואיננו מגדירים את מספר וסוג הפרמטרים שמועברים, הטיפול בפרמטרים זהה בכל הפונקציות. בגוף הפונקציה ניתן להתיחס לפרמטרים שהועברו באמצעות המשתנים המיוחדים $1, $2, $3 וכן הלאה (או לרשימת כל הפרמטרים באמצעות $@).   
בהפעלת הפונקציה אנו מעבירים את הארגומנטים לאחר שם הפונקציה, ללא שימוש בסוגריים. הקוד הבא לדוגמא יוצר תיקייה ובתוכה יוצר קובץ:

#!/usr/bin/env bash

function mk\_and\_cd

{

mkdir $1

cd $1

}

mk\_and\_cd testdir

touch testfile

החזרת ערך מפונקציה:

המילה השמורה return מחזירה ערך מפונקציה, כך שערך ההחזר נשמר במשתנה $?. באמצעות return ניתן להחזיר אך ורק ערכים מספריים, וכך נראה קוד לדוגמא:

function add() {

local result

(( result = $1 + $2 ))

return $result

}

add 10 20

# prints 30

echo $?

מה קורה אם רוצים להחזיר ערך שאינו מספרי (למשל מחרוזת)? במצב כזה אנו נאלצים לבצע רמאות קלה— במקום להחזיר ערך אנו מדפיסים אותו, ואת הפעלת הפונקציה מבצעים בתוך החלפת פקודה. כך נראה הקוד:

#!/usr/bin/env bash

function max\_str

{

MAX\_STR=""

for ARG do

if [[ ${#MAX\_STR} -lt ${#ARG} ]]

then

MAX\_STR=$ARG

fi

done

echo $MAX\_STR

}

longest=$(max\_str "foo" "bar" "buzzfeed" "hi")

echo $longest

## הצגת תפריט באמצעות select

מבנה לולאת select:

מטרתה של לולאת select הינה לאפשר בחירה מתוך רשימת אפשרויות— למשל רשימת קבצים, תוכניות או פרטי מידע אחרים. הלולאה חוסכת לנו את הצורך לצייר את התפריט ולהבין מה בחר המשתמש.

הנה המבנה הכללי של פקודת select:

select NAME in WORDS

do

COMMAND1

COMMAND2

COMMAND3

…

done

הלולאה מתחילה במילה השמורה select ולאחריה שם משתנה שיכיל את הבחירה (נקרא לו NAME). ניתן להמשיך לרשום את רשימת האפשרויות, או אם תשמיטו אותה Bash ישתמש ב $@. בתוך גוף הלולאה אפשר לקרוא את תוכן המשתנה $NAME  ולפיו להחליט על המשך הפעולה.  
בסיום בלוק הלולאה יודפס שוב התפריט. ניתן לצאת מהלולאה באמצעות פקודת break, או אם המשתמש הכניס תו סוף קלט.

דוגמא: הצגת תפריט:

נתחיל בדוגמא פשוטה המציגה תפריט, מדפיסה למשתמש את בחירתו ומסיימת:

#!/usr/bin/env bash

select choice in unix linux windows

do

echo you chose: $choice

break

done

דוגמא: בחירת קובץ:

התוכנית הבאה מבקשת מהמשתמש לבחור קובץ מהתיקייה, ומשנה את סיומת הקובץ לסיומת old:

#!/usr/bin/env bash

select FNAME in $(find . -maxdepth 1 -type f)

do

mv "${FNAME}" "${FNAME%.\*}.old"

done

דוגמא: בחירת סיומת:

התוכנית הבאה מבקשת מהמשתמש לבחור סיומת מתוך מגוון סיומות הקבצים בתיקיה, והתוכנית תדפיס כמה קבצים קיימים עם הסיומת שנבחרה:

#!/usr/bin/env bash

select FNAME in $(find . -iname '?\*.\*?' -maxdepth 1 -type f| cut -d . -f3 | sort -u )

do

find . -iname "\*.${FNAME}" -type f -maxdepth 1 | wc -l

done

## פענוח מתגים באמצעות getopts

תוכניות יוניקס רבות מופעלות משורת הפקודה באותו האופן: הן מקבלות מתגים המסומנים באמצעות מקף, ולאחר מכן אובייקטים לפעול עליהם. חלק מהמתגים אופציונאליים וחלק הכרחיים, חלקם מקבלים פרמטרים וחלקם בוליאנים. אנו גם נוהגים ביוניקס לאפשר חיבור בין מתגים, כך שניתן לרשום מספר אותיות אחרי סימן המקף.

פענוח מתגים שהועברו לסקריפט יכול להיות משימה לא פשוטה. רק חשבו על התוכנית ls, עם הפעלה לדוגמא שנראית כך:

$ ls -rt -l -S

אם גם הסקריפט שלכם צריך לקבל מספר מתגים, בלי להתחשב בסדר הופעתם ותוך אפשרות למשתמש לשלב את חלקם לרצף אחד, פענוח המחרוזת שקיבלתם יכול להיות מייגע.

הפקודה getopts הינה חלק מ Bash ותפקידה לפענח עבורכם רשימות מתגים. מאחר ורוב תוכניות יוניקס משתמשות באותו המבנה, שילוב getopts בסקריפט שלכם יגרום למשתמשים להרגיש בבית כשמשתמשים בתוכנית.

מינוחים:

ניקח את הפעלת התוכנית הבאה:

mybackup -x -f /etc/mybackup.conf -r ./foo.txt bar.txt

התוכנית מקבלת מספר מתגים ואובייקטים:

המתג -x הינו מתג בוליאני. התוכנית בודקת את נוכחותו או היעדרו.

המתגים -f ו -r מקבלים כל אחד פרמטר (במקרה זה שם קובץ). התוכנית מושפעת גם מנוכחותו של המתג אך גם מהפרמטר שהוא קיבל.

הטקסט bar.txt הינו אובייקט עליו התוכנית תעבוד.

כעת שאנו מבינים את המינוחים, נמשיך לפקודת getopts ונראה כיצד לפענח מתגים אלו.

פענוח מתגים:

הפקודה getopts מקבלת רשימה של מתגים אפשריים ומחזירה את המתג המפוענח הבא, או כשלון אם נגמרו המתגים. באופן זה היא משתלבת טוב עם לולאת while. הפקודה getopts פועלת על משתני הפרמטרים $1, $2, $3 וכן הלאה, אך אינה משנה אותם— לכן כשמפעילים אותה בלולאה עלינו להזיז את המשתנים בעצמנו ובכל איטרציה למחוק את המתגים ש getopts כבר פענחה.

כשמפעילים את getopts עלינו לספק לה מחרוזת המייצגת את המתגים שהתוכנית שלנו תומכת בהם, ואיזה מהמתגים צפוי לקבל פרמטר. המידע מועבר בתור רצף אותיות, כאשר סימן הנקודותיים אחרי אחת האותיות מציין כי זהו מתג המצפה לפרמטר.

עבור הפקודה שראינו בסעיף הקודם, נוכל להשתמש בהפעלה הבאה:

getopts xf:r:

המציינת מתג בוליאני -x ושני מתגים המצפים לפרמטרים -f ו -r.

שילוב של getopts בתוך סקריפט נראה כך:

#!/usr/bin/env bash

while getopts "xf:r:" opt; do

case $opt in

x)

echo "found -x !";;

f)

echo "found -f. argument was: $OPTARG";;

r)

echo "found -r. argument was: $OPTARG";;

esac

done

ברצוננו לכתוב סקריפט המציג הודעה למסך מספר פעמים. המתגים שהסקריפט צריך לתמוך בהם:  
-c מתג המקבל פרמטר ומציין כמה פעמים להדפיס את ההודעה (אם לא הופיע נציג 5 פעמים).  
-o מצג המקבל פרמטר ומציין לאיזה קובץ לכתוב את ההודעה (אם לא הופיע נכתוב למסך).  
-l מתג בוליאני, אם הופיע נדפיס הודעה עם סימוני כוכביות לפני ואחרי.

לאחר המתגים התוכנית תקבל את ההודעה להדפיס. במידה ולא הועברה הודעה התוכנית תציג הודעת ברירת מחדל.

פתרון:

#!/usr/bin/env bash

COUNT=5

MESSAGE="Hello World"

while getopts "c:o:l" opt; do

case $opt in

c)

COUNT=$OPTARG;;

o)

OUTFILE=$OPTARG;;

l)

PADDING=1;;

esac

done

shift $((OPTIND-1))

# One-line if:

# if $1 is not empty -> change message

[[ ! -z $1 ]] && MESSAGE="$\*"

# if PADDING -> pad message

[[ $PADDING = 1 ]] && MESSAGE="\*\*\*\*\*\* $MESSAGE \*\*\*\*\*\*"

# truncate OUTFILE if it's not empty

[[ ! -z $OUTFILE ]] && :> $OUTFILE

for (( i=0; i < COUNT; i++ )) do

if [[ -z $OUTFILE ]]

then

echo "$MESSAGE"

else

echo "$MESSAGE" >> $OUTFILE

fi

done

כמה נקודות לשים לב אליהן, לגבי getopts וגם בנוסף:

1. לאחר פענוח כל הפרמטרים, אנו משתמשים בפקודה shift כדי להסיר את המתגים מרשימת הפרמטרים. כך ניתן בהמשך להשתמש ב-$1, $2 ו $\* בלי שיופיעו ברשימה המתגים.
2. getopts מתקשרת עם שאר התוכנית באמצעות משתנים: אנו מגדירים משתנים וערכי ברירות מחדל בתחילת התוכנית, ובקוד הטיפול במתגים משנים את הערכים. במהלך התוכנית ניתן להשתמש במשתנים אלו.
3. מחיקת תוכן הקובץ בלבד יעילה יותר מאשר מחיקה ויצירה מחדש (וגם לפעמים אין לנו הרשאות למחוק קובץ אך יש הרשאה לכתוב אליו). ב Bash הפקודה : היא הפקודה הריקה, ולכן ניתן לרוקן קובץ על ידי ניתוב פלט הפקודה הריקה אליו.