ملخص شامل للغة Bash

1. مقدمة عن لغة Bash

- ما هي لغة Bash؟
Bash (وهي اختصار لـ Bourne Again SHell) هي مفسر أوامر وواجهة سطر أوامر (Command-Line Interface - CLI) شائعة للغاية في أنظمة التشغيل الشبيهة بيونكس (Unix-like)، مثل Linux و macOS.
تُعتبر Bash تطويراً لـ Bourne Shell (sh) الأصلي، حيث تضيف العديد من الميزات والتحسينات الهامة. تعمل Bash كوسيط بين المستخدم ونواة نظام التشغيل، مما يتيح للمستخدم تنفيذ الأوامر والبرامج وإدارة النظام بكفاءة.
- **تاريخ تطورها بإيجاز:**
  - **Bourne Shell (sh):** تم تطويره بواسطة ستيفن بورن في مختبرات بيل (Bell Labs) وصدر عام **1979** مع Unix Version 7.
  - **Bash (bash):** تم تطويره بواسطة بريان فوكس لصالح مشروع جنو (GNU Project) كبديل حر ومفتوح المصدر لـ sh. صدرت النسخة التجريبية الأولى (Beta) عام **1989**.
  - أصبحت Bash الصدفة **الافتراضية** في معظم توزيعات Linux و macOS (حتى إصدار Catalina الذي اعتمد Zsh افتراضيًا، لكن Bash لا تزال متاحة).
- أهميتها واستخداماتها الرئيسية:
تُعد Bash أساسية لإدارة أنظمة Linux/Unix والتفاعل معها. تشمل استخداماتها:
  - تنفيذ الأوامر وإدارة العمليات.
  - التنقل في نظام الملفات وإدارة الملفات والمجلدات.
  - كتابة **سكربتات** لأتمتة المهام المتكررة (Shell Scripting).
  - إدارة المستخدمين والأذونات.
  - تكوين بيئة النظام (Environment Configuration).
  - الوصول عن بعد إلى الأنظمة عبر بروتوكول SSH.

2. بنية أوامر Bash الأساسية

- أساسيات كتابة الأوامر:
تتبع الأوامر في Bash بنية عامة واضحة:
command [options] [arguments]
command #: الأمر (مثل ls)
options #: الخيارات (مثل -l)
arguments #: المعاملات (مثل /home/user)

  - command: اسم الأمر أو البرنامج (مثل ls, cd, echo).
  - options (اختياري): تُعدّل سلوك الأمر، تبدأ عادةً بشرطة (-) أو شرطتين (--) (مثل ls -l, grep --color).
  - arguments (اختياري): البيانات أو الملفات التي يعمل عليها الأمر (مثل cp file1.txt file2.txt).
- **الأوامر الداخلية والخارجية:**
  - **الأوامر الداخلية (Built-in):** مدمجة في Bash نفسها (مثل cd, echo, pwd, export, read). **أسرع**

في التنفيذ لأنها لا تتطلب تحميل برنامج خارجي.
- **الأوامر الخارجية (External):** برامج منفصلة في نظام الملفات (عادة في /bin, /usr/bin). يتم البحث عنها في متغير البيئة PATH (مثل ls, grep, find, nano).

- **أمثلة بسيطة:**

```
# عرض محتويات المجلد الحالي (أمر خارجي)
ls

# عرض المحتويات بتنسيق طويل وتفصيلي (أمر خارجي مع خيار)
ls -l

# تغيير المجلد الحالي إلى المجلد الرئيسي للمستخدم (أمر داخلي)
cd ~

# طباعة نص بسيط على الشاشة (أمر داخلي مع معامل)
echo "مرحباً بالعالم!" # Hello World!
```

3. المتغيرات في Bash

- تعريف المتغيرات وأنواعها:

المتغيرات هي حاويات لتخزين البيانات. Bash لا تتطلب تعريف نوع المتغير صراحةً؛ يتم التعامل معها كنصوص (strings) بشكل عام، ولكن يمكن إجراء عمليات حسابية عليها.

- **طريقة تعيين واستخدام المتغيرات:**
- **التعيين:** باستخدام علامة (=) **بدون مسافات** حولها.

```
my_variable="Some text"  # متغير نصي
user_count=10            # متغير رقمي (يُعامل كنص مبدئيًا)
```

- **الاستخدام (الإحلال):** باستخدام علامة الدولار ($) قبل اسم المتغير.

```
echo $my_variable        # طباعة قيمة المتغير النصي
echo "عدد المستخدمين: $user_count" # طباعة قيمة المتغير الرقمي
echo "المسار الحالي: $PWD" # $PWD هو متغير بيئة جاهز
```

- يمكن استخدام الأقواس المتعرجة {} لتحديد اسم المتغير بوضوح، خاصة عند دمجه مع نصوص أخرى أو لتجنب الالتباس:

```
file_prefix="backup" # بادئة اسم الملف
# إنشاء اسم ملف يحتوي على البادئة وتاريخ اليوم
echo "${file_prefix}_$(date +%Y%m%d).log" # مثال للناتج: backup_20250408.log
```

- المتغيرات البيئية (Environment Variables):

متغيرات خاصة متاحة لجميع العمليات في الصدفة الحالية والعمليات الفرعية. تحدد بيئة عمل النظام والمستخدم.

- **أمثلة شائعة:**
  - PATH: مسارات البحث عن الأوامر الخارجية.
  - HOME: المجلد الرئيسي للمستخدم الحالي.
  - USER: اسم المستخدم الحالي.
  - PWD: المجلد الحالي (مسار العمل الحالي).
  - SHELL: مسار الصدفة (المفسر) المستخدمة.
- عرض جميع المتغيرات البيئية: env أو printenv.
- تعيين متغير بيئة (جعله متاحًا للعمليات الفرعية): .export

```
"MY_CUSTOM_VAR="some_value" # تعريف متغير عادي
export MY_CUSTOM_VAR       # تصدير المتغير ليصبح متغير بيئة

# أو يمكن التعريف والتصدير في خطوة واحدة
"export ANOTHER_VAR="another_value
```

4. التحكم في التدفق (Flow Control)

تُستخدم لتوجيه تنفيذ الأوامر بناءً على شروط أو لتكرار التنفيذ.

- **if/elif/else**: تنفيذ أوامر بناءً على تحقق شرط.

```
count=5 # تعيين قيمة للمتغير
if [ $count -gt 10 ]; then # إذا كان العدد أكبر من 10 (-gt: greater than)
  echo "العدد أكبر من 10"
elif [ $count -eq 10 ]; then # وإلا إذا كان العدد يساوي 10 (-eq: equal)
  echo "العدد يساوي 10"
else # وإلا (إذا لم تتحقق الشروط السابقة)
  echo "العدد أصغر من 10"
fi # نهاية كتلة الأوامر الخاصة بـ if
```

- [أو test تُستخدم لتقييم الشروط. أمثلة شائعة للشروط: -gt (أكبر من)، -eq (يساوي)، -lt (أصغر من)، -ne (لا يساوي)، -ge (أكبر أو يساوي)، -le (أصغر أو يساوي)، -f (ملف موجود وهو ملف عادي)، -d (ملف موجود وهو مجلد)، -z (النص فارغ)، -n (النص غير فارغ).
- **for**: تكرار تنفيذ أوامر على مجموعة من العناصر أو لعدد محدد.

```
# التكرار على قائمة محددة من النصوص
for fruit in "apple" "banana" "orange"; do # fruit هو متغير التكرار
  echo "أنا أحب $[fruit_name] # I love fruit"
done # نهاية الحلقة

# التكرار باستخدام تسلسل أرقام (من 1 إلى 5)
for i in {1..5}; do
```

```
    echo "الرقم: $[i]" # Number: [i]
done

# التكرار على نتائج أمر (مثل أسماء الملفات النصية في المجلد الحالي)
for filename in *.txt; do # filename هو متغير التكرار
    # التحقق أولاً إذا كان الملف موجودًا بالفعل (لتجنب الخطأ إذا لم توجد ملفات txt)
    if [ -f "$filename" ]; then
        echo "معالجة الملف: $[filename]" # Processing file: [filename]
        # هنا يمكن إضافة أوامر أخرى لمعالجة كل ملف
    fi
done
```

- **while**: تكرار طالما أن شرطًا معينًا **متحقق** (true).

```
counter=1 # تهيئة العداد
while [ $counter -le 5 ]; do # طالما أن العداد أصغر من أو يساوي 5 (-le: less than or equal)
    echo "العداد الحالي: $[counter]" # Current counter: [counter]
    # زيادة العداد لتجنب حلقة لا نهائية!
    ((counter++)) # طريقة Bash لزيادة المتغير بواحد
    # أو: counter=$((counter + 1)) # طريقة أخرى للعمليات الحسابية
done # نهاية الحلقة
```

- **until**: تكرار **حتى** يصبح شرط معين متحققًا (true) - عكس while.

```
counter=1 # تهيئة العداد
until [ $counter -gt 5 ]; do # كرر حتى يصبح العداد أكبر من 5 (-gt: greater than)
    echo "العداد (until): $[counter]" # Counter (until): $counter
    ((counter++)) # زيادة العداد
done # نهاية الحلقة
```

- **case**: اختيار تنفيذ أوامر بناءً على تطابق قيمة مع نمط. بديل جيد لجمل if/elif/else المتعددة والمعقدة.

```
# قراءة إدخال من المستخدم وتخزينه في المتغير user_choice
read -p "أدخل خياراً (yes/no/maybe): " user_choice # Enter an option (yes/no/maybe):

case $user_choice in # ابدأ التحقق من قيمة user_choice
    yes|YES|y ) # إذا كانت القيمة yes أو YES أو y
        echo "لقد اخترت نعم." # You chose yes.
        ;; # نهاية هذه الحالة (ضرورية)
    (no|NO|n) # إذا كانت القيمة no أو NO أو n
        echo "لقد اخترت لا." # You chose no.
```

```
;;
maybe ) # إذا كانت القيمة maybe
echo "لقد اخترت ربما." # You chose maybe.
;;
*) # الحالة الافتراضية (إذا لم تتطابق أي من الحالات السابقة)
echo "خيار غير صالح." # Invalid option.
;;
esac # نهاية جملة case (كلمة case معكوسة)
```

5. الدوال (Functions) في Bash

تسمح بتجميع مجموعة أوامر تحت اسم واحد لإعادة الاستخدام وتنظيم الشيفرة.

- **تعريف الدوال وإنشائها واستدعائها:**

```
# طريقة التعريف 1 (باستخدام كلمة function)
function greet { # greet هو اسم الدالة
echo "مرحباً، $1!" # $1 هو المعامل الأول الممرر للدالة
# Hello, [first argument]!
}

# طريقة التعريف 2 (الأكثر شيوعًا وبدون كلمة function)
welcome() { # welcome هو اسم الدالة
local name=$1 # استخدام local لتعريف متغير محلي (نطاقه داخل الدالة فقط)
local city=$2 # $2 هو المعامل الثاني
echo "أهلاً بك يا $name من مدينة $city] from [name] Welcome # ".[city.
}

# --- استدعاء الدوال ---
greet "أحمد" # تمرير "أحمد" كمعامل أول ($1) للدالة greet
welcome "سارة" "الرياض" # تمرير "سارة" كـ $1 و "الرياض" كـ $2 للدالة welcome
```

- **نطاق المتغيرات داخل الدوال:**
  - **Global (عام):** افتراضيًا، المتغيرات المعرفة داخل الدالة تكون عامة ويمكن الوصول إليها وتعديلها من خارج الدالة.
  - **Local (محلي):** يُفضل **دائمًا** استخدام الكلمة المفتاحية local لتعريف متغيرات يقتصر نطاقها على الدالة فقط. هذا يمنع التعديلات غير المقصودة على متغيرات بنفس الاسم خارج الدالة ويجعل الشيفرة أوضح وأسهل للصيانة.
- **تمرير المعاملات وإرجاع القيم:**
  - **المعاملات (Arguments):** تُمرر عند استدعاء الدالة وتكون متاحة داخلها عبر المتغيرات الموضعية:
    - $1, $2, $3, ... : المعامل الأول، الثاني، الثالث، إلخ.

- $0: اسم السكربت أو الدالة نفسها.
- $#: عدد المعاملات الممررة.
- $*: جميع المعاملات كسلسلة نصية واحدة.
- $@: جميع المعاملات ككلمات منفصلة (مفيد عند التمرير لأوامر أخرى).

- **إرجاع القيم (Bash Return Values):** لا "ترجع" قيمة بالمعنى التقليدي. بدلاً من ذلك:

1. **إرجاع حالة خروج (Exit Status):** رقم بين 0 (نجاح) و 255 (خطأ) باستخدام أمر return. يمكن التقاط هذه الحالة باستخدام المتغير الخاص $? **فورًا** بعد استدعاء الدالة.
2. **طباعة الناتج إلى الخرج القياسي (stdout):** يمكن التقاط هذا الناتج باستخدام **إحلال الأوامر** $(...). هذه هي الطريقة الشائعة لمحاكاة إرجاع قيمة نصية أو رقمية.

```
# مثال على حالة الخروج (للتحقق من النجاح أو الفشل)
check_file() {
  if [ -f "$1" ]; then # هل الملف ($1) موجود وهو ملف عادي؟
    echo "الدالة check_file: موجود '$1' الملف." # Function check_file: File '[filename]' exists.
    return 0 # إرجاع 0 يعني نجاح
  else
    echo "الدالة check_file: غير موجود '$1' الملف." # Function check_file: File '[filename]' not found.
    return 1 # إرجاع قيمة غير صفرية (عادة 1) يعني خطأ
  fi
}

check_file "myfile.txt" # استدعاء الدالة
exit_status=$? # تخزين حالة الخروج فورًا في متغير
echo "حالة الخروج للدالة check_file هي: $exit_status" # Exit status of check_file is: [status]
if [ $exit_status -eq 0 ]; then
  echo "النتيجة: الملف موجود." # Result: File exists.
else
  echo "النتيجة: الملف غير موجود." # Result: File not found.
fi

# مثال على التقاط الخرج (للحصول على قيمة)
get_sum() {
  local num1=$1
  local num2=$2
  local sum=$((num1 + num2)) # إجراء عملية حسابية
  echo $sum # طباعة الناتج إلى stdout (هذا ما سيتم التقاطه)
}

result=$(get_sum 5 3) # استدعاء الدالة والتقاط خرجها في المتغير result
echo "ناتج الجمع هو: $result" # The sum is: 8
```

6. عمليات الإدخال والإخراج (I/O Redirection)

تغيير مصدر الإدخال القياسي (stdin - لوحة المفاتيح عادةً) أو وجهة الخرج القياسي (stdout - الشاشة عادةً) وخرج الخطأ القياسي (stderr - الشاشة عادةً).

- **>** (توجيه الخرج - الكتابة فوق الملف): يوجه stdout إلى ملف. **يحذف محتوى الملف القديم إن وجد.**
```
ls -l > file_list.txt # حفظ قائمة الملفات في file_list.txt (الكتابة فوق الملف)
echo "سطر جديد" > existing_file.txt # الكتابة فوق محتوى الملف existing_file.txt
echo "New line" # "
```

- **>>** (توجيه الخرج - الإلحاق بالملف): يوجه stdout إلى ملف. **يضيف الخرج إلى نهاية الملف** إن وجد، أو ينشئ الملف.
```
date >> system.log # إضافة التاريخ والوقت الحالي إلى نهاية ملف system.log
echo "سطر آخر" >> existing_file.txt # إضافة هذا السطر لنهاية الملف
"Another line" #
```

- **<** (توجيه الإدخال): يوجه محتوى ملف ليكون stdin لأمر.
```
sort < unsorted_list.txt # قراءة القائمة من الملف unsorted_list.txt وفرزها
wc -l < data.csv # عد الأسطر في ملف data.csv باستخدام wc -l
```

- **|** (Pipe - الأنبوب): يوجه stdout لأمر ليكون stdin لأمر آخر. لربط الأوامر معًا.
```
# عرض الملفات بتنسيق طويل ثم تمرير الخرج لـ grep للبحث عن الأسطر التي تحتوي "."txt"
ls -l | grep ".txt"

# سلسلة أوامر لتحليل ملف سجل الوصول:
# 1. عرض محتوى الملف
# 2. قص العمود الأول (عادة عنوان IP) باستخدام المسافة كفاصل
# 3. فرز العناوين
# 4. عد تكرار كل عنوان فريد
cat access.log | cut -d ' ' -f 1 | sort | uniq -c
```

- **توجيه خرج الخطأ (stderr - File Descriptor 2):**
```
# البحث عن ملف، توجيه النتائج (stdout) لملف والأخطاء (stderr) لملف آخر
find / -name "secret" > results.txt 2> errors.log

# توجيه stdout و stderr لنفس الملف (الطريقة التقليدية)
find / -name "secret" > all_output.txt 2>&1

# توجيه stdout و stderr لنفس الملف (اختصار في Bash الحديثة)
find / -name "secret" &> combined_output.txt
```

7. التعامل مع الملفات والمجلدات

Bash توفر مجموعة غنية من الأوامر لإدارة نظام الملفات.

- **أوامر أساسية للملفات والمجلدات:**
  - ls: عرض محتويات المجلد (ls -l للتفاصيل، ls -a للمخفي، ls -lh لحجم مقروء).
  - cd: تغيير المجلد الحالي (cd .. للأصل، cd ~ أو cd للمنزل، cd - للمجلد السابق).
  - pwd: طباعة مسار المجلد الحالي (Print Working Directory).
  - mkdir: إنشاء مجلد جديد (mkdir mydir, mkdir -p path/to/nested/dir لإنشاء مسار كامل).
  - rmdir: حذف مجلد **فارغ**.
  - rm: حذف ملفات أو مجلدات (rm file.txt, rm -r directory لحذف مجلد ومحتوياته **بحذر شديد!**، rm -f للحذف بقوة بدون تأكيد، rm -i للحذف مع تأكيد).
  - cp: نسخ ملفات أو مجلدات (cp source.txt destination.txt, cp file.txt dir/, cp -r /source_dir/ destination_dir).
  - mv: نقل أو إعادة تسمية (mv old.txt new.txt, mv file.txt target_dir/).
  - touch: إنشاء ملف فارغ أو تحديث وقت تعديل/وصول ملف موجود (touch newfile.txt).
- **أوامر أساسية لمحتوى الملفات:**
  - cat: عرض محتوى ملف بالكامل (cat file.txt).
  - less / more: عرض محتوى ملف صفحة بصفحة (less أحدث وأكثر مرونة).
  - head: عرض الأسطر الأولى (الافتراضي 10، head -n 5 file.txt لأول 5).
  - tail: عرض الأسطر الأخيرة (الافتراضي 10، tail -n 20 file.txt لآخر 20، tail -f log.txt للمراقبة الحية).
  - grep: البحث عن نمط نصي (grep "error" system.log, ls -l | grep "Apr").
  - wc: عد الأسطر (-l)، الكلمات (-w)، الحروف (-c) (wc -l file.txt).
  - sort: فرز أسطر ملف (أبجديًا افتراضيًا، sort -n للفرز الرقمي).
  - uniq: إزالة الأسطر المكررة **المتجاورة** (غالبًا تستخدم بعد sort). (sort file.txt | uniq -c لعد التكرارات).
- الأذونات (Permissions):

تحدد من يمكنه القراءة (r)، الكتابة (w)، والتنفيذ (x) للملفات والمجلدات. تنقسم لثلاث فئات: المالك (user)، المجموعة (group)، الآخرون (others).

  - ls -l: يعرض الأذونات (مثل -rwxr-xr--).
  - chmod: تغيير الأذونات (مثل chmod +x script.sh لإضافة إذن التنفيذ للمالك والمجموعة والآخرين، chmod u+x script.sh للمالك فقط، chmod 755 file.txt لتعيين rwxr-xr-x).
  - chown: تغيير المالك (chown new_owner file.txt).
  - chgrp: تغيير المجموعة (chgrp new_group file.txt).

8. التعابير النمطية (Regular Expressions) واستخدامها مع Bash

- مقدمة بسيطة:

التعابير النمطية (Regex) هي لغة لوصف الأنماط في النصوص. أداة قوية جدًا للمطابقة والبحث والاستبدال.

  - **أمثلة على رموز Regex الأساسية (BRE - Basic Regular Expressions):**

- .: يطابق أي حرف واحد (ما عدا سطر جديد).
- *: يطابق صفر أو أكثر من الحرف/المجموعة السابقة.
- ^: يطابق بداية السطر.
- $: يطابق نهاية السطر.
- []: يطابق أي حرف واحد داخل الأقواس (مثل [aeiou]).
- [^]: يطابق أي حرف واحد ليس داخل الأقواس.
- \: لإلغاء المعنى الخاص للحرف التالي (مثل \. لمطابقة النقطة حرفيًا).

- **Regex الموسعة (ERE - Extended Regular Expressions):** تُفعّل غالبًا بخيار مثل -E مع grep. تضيف رموزًا مثل:
  - +: يطابق واحد أو أكثر من السابق.
  - ?: يطابق صفر أو واحد من السابق.
  - |: يطابق التعبير على اليسار **أو** التعبير على اليمين.
  - (): لتجميع أجزاء من التعبير.

- **أمثلة على الاستخدام:**

- **grep**: للبحث عن الأسطر التي تطابق نمطًا.

```
# البحث عن الأسطر التي تبدأ بكلمة "Error:" (حساس لحالة الأحرف)
grep "^Error:" logfile.txt

# البحث عن الأسطر التي تحتوي على أرقام فقط (باستخدام Regex الموسعة ERE)
grep -E "^[0-9]+$" data.txt

# البحث عن عناوين البريد الإلكتروني (مثال مبسط جدًا) واستخراجها فقط (-o)
grep -E -o "\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b" textfile.txt
```

- **sed**: محرر تدفق لمعالجة النصوص (خاصة الاستبدال والحذف).

```
# استبدال أول ظهور لكلمة "old" بـ "new" في كل سطر
sed 's/old/new/' input.txt

# استبدال كل ظهور لكلمة "old" بـ "new" في كل سطر (g = global)
sed 's/old/new/g' input.txt

# حذف الأسطر التي تحتوي على كلمة "debug"
sed '/debug/d' input.txt
```

- **awk**: أداة قوية لمعالجة النصوص والبيانات المنظمة في أعمدة (الحقول).

```
# طباعة العمود الأول ($1) والثالث ($3) من ملف مفصول بمسافات (الفاصل الافتراضي)
awk '{print $1, $3}' data.txt

# طباعة الأسطر التي يكون فيها العمود الثاني ($2) أكبر من 10
```

awk '$2 > 10 {print $0}' scores.txt # $0 يمثل السطر بأكمله

9. كتابة السكربتات (Shell Scripting)

السكربت هو ملف نصي يحتوي سلسلة أوامر Bash لتنفيذ مهمة معينة.

- **أساسيات كتابة سكربت Bash:**
  - **Shebang (#!):** السطر الأول **يجب** أن يكون #!/bin/bash (أو مسار Bash الصحيح). يخبر النظام باستخدام Bash لتنفيذ السكربت.
  - السكربت يحتوي أوامر Bash، متغيرات، هياكل تحكم، دوال، وتعليقات (#).

```
#!/bin/bash

# ============================
# سكربت ترحيبي بسيط
# يعرض رسالة واسم المستخدم والتاريخ
# ============================

# الحصول على اسم المستخدم الحالي وتخزينه
USERNAME=$(whoami) # استخدام إحلال الأوامر $(...)

# طباعة رسالة الترحيب
echo "مرحباً بك في عالم السكربتات يا $USERNAME!" # Welcome to the world of scripting, [username]!

# طباعة التاريخ والوقت الحاليين
echo "تاريخ ووقت اليوم هو: $(date)" # Today's date and time is: [date output]

# إنهاء السكربت بحالة نجاح (0). هذا اختياري في نهاية السكربت.
exit 0
```

- **تنفيذ السكربتات ومنحها صلاحيات التنفيذ:**

1. **منح إذن التنفيذ (Execute Permission):** لمرة واحدة فقط لكل ملف سكربت.

```
chmod +x myscript.sh # إضافة (+) إذن التنفيذ (x)
# أو (rwxr-xr-x) chmod 755 myscript.sh)
```

2. **التنفيذ:**

- إذا كان السكربت في المجلد الحالي: ./myscript.sh (النقطة والشرطة المائلة ضروريتان)
- إذا كان مسار السكربت ضمن متغير PATH: myscript.sh (مثل الأوامر العادية)
- باستخدام المفسر مباشرة (لا يتطلب إذن التنفيذ): bash myscript.sh

- **أفضل الممارسات (Best Practices):**
  - ابدأ دائمًا بـ #!/bin/bash.
  - **أضف تعليقات** لشرح المنطق (# comment).

- استخدم أسماء متغيرات ودوال **واضحة ووصفية** (مثل user_count بدلاً من uc).
- استخدم local للمتغيرات داخل الدوال.
- **ضع المتغيرات النصية والمسارات بين علامتي اقتباس مزدوجة** ("$variable") لتجنب المشاكل مع المسافات والحروف الخاصة.
- تحقق من **حالة الخروج ($?)** للأوامر المهمة لمعالجة الأخطاء.
- فكر في استخدام set -e (للخروج عند أول خطأ)، set -u (للخروج عند استخدام متغير غير معرف)، و set -o pipefail (لفشل الأنابيب إذا فشل أي جزء) في بداية السكربتات الأكثر تعقيدًا (مع فهم تأثيرها).
- اجعل السكربتات **قابلة لإعادة الاستخدام** ومرنة قدر الإمكان (مثلاً باستخدام معاملات بدل القيم الثابتة).

10. مفاهيم متقدمة (اختياري)

- **Here Documents (<<DELIMITER):** طريقة لتمرير عدة أسطر من الإدخال (stdin) إلى أمر مباشرة داخل السكربت.

```
# تمرير نص متعدد الأسطر إلى أمر cat
cat << EOF
هذا هو السطر الأول من النص. # This is the first line.
وهذا هو السطر الثاني.      # This is the second line.
المتغير USER هو: $[username] :is The USER variable # The USER
EOF
EOF # هو محدد النهاية (Delimiter)، يمكن أن يكون أي كلمة (عادةً بأحرف كبيرة)
# يجب أن يكون محدد النهاية في سطر منفصل بمفرده وبدون مسافات بادئة.

# مثال آخر مع أمر تفاعلي مثل ftp
ftp -n ftp.example.com << FTP_COMMANDS
user anonymous myemail@example.com # تسجيل الدخول كمستخدم مجهول
cd /pub                            # تغيير المجلد إلى /pub
get somefile.zip                   # تحميل ملف
quit                               # تسجيل الخروج
FTP_COMMANDS
```

- **Process Substitution (<() و >()):** يسمح بمعاملة خرج أمر كما لو كان ملفًا (للقراءة)، أو إرسال خرج إلى عملية كما لو كانت ملفًا (للكتابة). يتجنب الحاجة لملفات مؤقتة.

```
# مقارنة خرج أمرين (بعد فرزهما) بدون الحاجة لملفات مؤقتة
diff <(sort file1.txt) <(sort file2.txt)
command)> #) يُنفذ الأمر ويعطي مسارًا مؤقتًا لخرجه يمكن قراءته كملف

# مثال معقد: إرسال خرج ls -l إلى tee، والذي بدوره يرسله لثلاث وجهات:
# 1. إلى wc -l (عبر >(command)) لعد الأسطر
# 2. إلى grep (عبر >(command)) لتصفية وحفظ ملفات txt
3 #. إلى less (عبر الأنبوب |) للعرض على الشاشة
```

```
ls -l | tee >(wc -l > /dev/stderr) >(grep ".txt" > txt_files.log) | less
command)> # ) يُنفذ الأمر ويجعل مدخله متاحًا للكتابة إليه كملف
# تم توجيه خرج wc إلى stderr هنا فقط لتوضيح أنه يعمل بشكل منفصل
```

- **Arrays (المصفوفات):** لتخزين قائمة من القيم في متغير واحد. الفهرس يبدأ من 0.

```
# --- تعريف مصفوفة ---
# الطريقة 1:
"my_array=("apple" "banana" "cherry") # عناصر نصية
# الطريقة 2: باستخدام نتائج أمر
files=(*.log) # إنشاء مصفوفة بأسماء ملفات .log في المجلد الحالي

# --- الوصول للعناصر ---
echo ${my_array[0]} # طباعة العنصر الأول (apple)
echo ${my_array[1]} # طباعة العنصر الثاني (banana)

# --- الوصول لجميع العناصر ---
"echo "All items (*): ${my_array[*]}" # يعامل كل العناصر ككلمة واحدة
"echo "All items (@): ${my_array[@]}" # يعامل كل عنصر ككلمة منفصلة (أكثر استخدامًا)
# الفرق يظهر بوضوح عند استخدام علامات الاقتباس:
# for item in "${my_array[*]}"; do ... done # حلقة واحدة فقط
# for item in "${my_array[@]}"; do ... done # حلقة لكل عنصر (مفضل)

# --- عدد العناصر ---
"echo "Number of items: ${#my_array[@]}" # طباعة عدد العناصر (3 في المثال الأول)

# --- إضافة عنصر ---
"my_array+=("orange") # إضافة عنصر جديد لنهاية المصفوفة
echo "After adding: ${my_array[@]}" # apple banana cherry orange

# --- التكرار على عناصر المصفوفة (الطريقة المفضلة) ---
for item in "${my_array[@]}"; do
echo "Item: $item" # Element: [item]
done
```

تم إنتاج هذا الملخص بواسطة المبرمج SayyadN.