IT資格道場

LPIC-1 102試験 学習用テキスト

予備知識ゼロから合格ラインへ —— 読んで理解する1冊目

無料サンプル(第1章まで)

LPI公式 LPIC-1 Exam 102 Objectives Version 5.0 準拠
2026年7月版

本書のご利用にあたって

- 本書は「IT資格道場」(www.shikaku-doujou.com)の購入者限定テキストのうち、学習用テキストの第1章までを収録した**無料サンプル**です。どなたでもご覧いただけます。
- 無料サンプルであっても著作物です。本書の転載・改変しての再配布・二次販売を禁じます。
- 本書を AI モデルの学習データとして利用すること(学習目的の入力・データセット化を含む)を禁じます。
- 製品版(学習用テキスト・頻出度別ノート・暗記用ノートの3点)は、ご購入者を識別できる透かし入りのPDFとして提供されます。
- 本PDFはスマートフォン・タブレットでの閲覧に合わせたA5版面・文字サイズで組版しています。

不正な配布を見つけた場合は info@shikaku-doujou.com までご連絡ください。

目次

この教材群の使い方(3点セット)

種類	役割
① 学習用テキスト(本書)	これだけで問題が解けるようになる本編
② 頻出度別ノート	テキストを読んだら次はこれ。頻出順に絞った問題と解説で「解ける」に橋渡し。試験直前の最終チェックにも
③ 暗記用ノート	覚えるべき要点だけを一問一答に凝縮。空き時間の反復と用語の再確認用

使い方: ① 学習用を読む → ② 頻出度別で解き方を掴む → ③ 問題演習で腕試し → ④ 頻出度別に戻って再確認(試験直前もここ)。暗記用は空き時間や用語の再確認に。

このテキストの位置づけ: 本書は①の学習用テキストです。まずはここを通読し、これだけで問題が解けるようになることを目標にしてください。

はじめに —— この教材の使い方

このテキストは、Linux Professional Institute（LPI）が実施する **LPIC-1 102 試験（試験コード 102-500）** に合格することを目的に作られています。

102試験が問うのは、Linux を「運用する側」の土台です。シェルとスクリプトで作業を自動化し（Topic 105）、デスクトップとアクセシビリティを整え（Topic 106）、ユーザー・ジョブ・ロケールを管理し（Topic 107）、時刻・ログ・メール・印刷という必須サービスを動かし（Topic 108）、ネットワークをつないで切り分け（Topic 109）、ホストとデータを守る（Topic 110）—— この6つの Topic に、公式の到達目標 19 本が並びます。

覚え方は3点に絞ります。

1. **似ているコマンドの役割の違い**（例: `useradd` と `adduser`、`cron` と `at`、`ss` と `netstat`）
2. **書式とオプションを「打てる」まで**（本試験は択一に加えて、コマンド名やパスを直接入力する記述入力があります。本書は主要コマンドに書式行と代表オプション表を必ず置きます）
3. **設定が「今だけ」効くのか「次回起動後も」効くのか**（例: `export` と `~/.bashrc`、`date -s` と `hwclock -w`、`ip addr add` と設定ファイル）

試験の基本情報

項目	内容
試験名	LPIC-1 102試験（**102-500**）
運営	Linux Professional Institute（LPI）
出題範囲のバージョン	**Version 5.0**
認定要件	101試験と102試験の両方に合格すると LPIC-1 に認定される。前提資格は不要
問題数	**60問**
出題形式	**択一（multiple-choice）と記述入力（fill-in-the-blank）**
試験時間	**90分**
合格ライン	**200〜800 のスケールで 500点**。必要な正答数は出題の組み合わせによって変わる

試験の設計図 —— 4つの Topic と weight

公式は到達目標ごとに **weight**（重み）を公表しており、「weight n の到達目標は本番で n 問出る」と明記しています。合計は 60 です。本書の章立てはこの Topic の順番どおり（第1章〜第6章）で、章の分量も weight に比例させています。

Topic	内容	weight	本書の章
105	シェル、スクリプト	8	第1章
106	ユーザーインターフェースとデスクトップ	4	第2章
107	管理タスク	12	第3章
108	必須システムサービス	12	第4章
109	ネットワークの基礎	14	第5章
110	セキュリティ	10	第6章

4 ステップ学習法

1. 第1章から順に読む（Topic 109 と 107・108 が山です）
2. 各節の「書式行」と「取り違えやすい対」の表を、手を動かして確かめる
3. 問題演習で腕試しをし、間違えた問題の該当節に戻る
4. 頻出度別ノートで直前の総仕上げ

第1章 シェル、スクリプト(Topic 105・weight 8)

この章は公式の副目標 **105.1 シェル環境をカスタマイズして使用する(weight 4)** と **105.2 簡単なスクリプトをカスタマイズする(weight 4)** に当たります。

シェルは、あなたが打った文字を解釈してカーネルに実行させる「通訳」です。通訳がどう振る舞うかは、**起動のされ方によって読み込む設定ファイルが変わる**ことで決まります。「設定を書いたのに反映されない」という実務のつまずきも、試験の設問も、ほとんどがこの一点から生まれます。

前半(1-1)は、その設定ファイルの読み込み順序と、変数・検索パス・エイリアス・関数という「環境を作る道具」を扱います。後半(1-2)は、その環境の上で動く**シェルスクリプト**を、書けて読めて権限まで管理できる状態にします。

この試験は択一だけでなく**記述入力(fill-in-the-blank)**が公式に採用されている形式です。コマンド名・ファイル名・オプションは、見て選べるだけでなく**その綴りのまま打てる**ところまで持っていってください。本章では主要なコマンドに書式行と代表オプションの表を付けています。

1-1 シェル環境をカスタマイズして使用する [105.1]

到達目標は「利用者の要求に合わせてシェル環境をカスタマイズできること」「全体プロファイルと個人プロファイルを変更できること」です。出題の芯は 3 つ —— **どのファイルがいつ読まれるか、変数が子プロセスへ渡るか渡らないか、変更を今のシェルに効かせる方法**です。

1-1-1 シェルとログインシェル —— 4 つの性格 [105.1]

シェル(shell)とは、利用者が入力したコマンドを解釈して OS に実行させるプログラムです。Linux で標準的に使われるのが **bash**(Bourne Again Shell)で、LPIC-1 の出題も bash が前提です。利用できるログインシェルの一覧は `/etc/shells` にあり、ユーザーごとの既定シェルは `/etc/passwd` の 7 番目のフィールドに書かれています。

シェルは**起動のされ方**で性格が変わり、その性格によって**読み込む設定ファイルが変わります**。ここが 105.1 の核心です。

性格	どういうときか	代表例
ログインシェル	ログイン操作を経て起動された	コンソールログイン、`ssh` でのリモートログイン、`su -`、`bash --login`
非ログインシェル	すでにログインした状態から起動された	デスクトップで端末アプリを開く、`bash` と打って入れ子で起動、`su`(ハイフン無し)
対話的シェル	人間がキーボードから打つ前提	端末に出ているプロンプト
非対話的シェル	人間の入力を前提としない	スクリプトを `bash script.sh` で実行したとき、cron から実行されたとき

「ログインしたか」と「対話的か」は**別の軸**です。ログインシェルであって対話的、非ログインであって対話的、非対話的でログイン(`bash --login script.sh`)、といった組み合わせがすべて成立します。設問は「ログインシェル/非ログインシェル」の軸で聞いてくることが多いので、まずこの軸を確定させてから設定ファイルを思い出してください。

いま自分がどちらにいるかは次で調べられます。

- `echo $0` —— ログインシェルでは先頭にハイフンが付いて **`-bash`** と表示される(非ログインなら `bash`)
- `shopt -q login_shell; echo $?` —— ログインシェルなら `0`
- `echo $SHELL` は**既定のログインシェル**を示すだけで、いま動いているシェルとは限りません(`/etc/passwd` 由来の値がそのまま入っているため)

⚠️ `su` と `su -` の違いはそのまま設問になります。**`su -` (または `su -l` ・`su --login`)はログインシェルとして起動する**ので、移動先ユーザーのプロファイルが読まれ、環境変数も作業ディレクトリも切り替わります。ハイフン無しの `su` は非ログインシェルなので、**元のユーザーの環境変数を引きずったまま**になります。「root になったのに検索パスに `/sbin` が入っていない」というトラブルの典型です。

シェルの 2 つの軸 —— 「ログインしたか」と「対話的か」は別の軸

	ログインシェル	非ログインシェル
対話的シェル	コンソールログイン ssh でのリモートログイン `su -`	端末アプリを開く bash と打って入れ子で起動 su(ハイフン無し)
非対話的シェル	`bash --login script.sh` で実行したとき	bash script.sh で実行 cron から実行されたとき

echo $0 —— ログインシェルでは先頭にハイフンが付いて -bash と表示される
shopt -q login_shell; echo $? —— ログインシェルなら 0
su - はログインシェルとして起動するので、移動先ユーザーのプロファイルが読まれる

図 1-1　シェルの 2 つの軸 ——「ログインしたか」と「対話的か」は別の軸

1-1-2 起動時に読み込まれる設定ファイルと読み込み順 [105.1]

ログインシェルとして起動した場合

1. `/etc/profile` —— 全ユーザー共通の設定。**最初に読まれます**
2. 続いて個人の設定を `~/.bash_profile` → `~/.bash_login` → `~/.profile` の順に**探し**ます

ここが最頻出のひっかけです。**個人設定の 3 つは、最初に見つかった 1 つだけが読み込まれます。**3 つとも存在していても実行されるのは `~/.bash_profile` だけで、残り 2 つは読まれません。「3 つが順に全部読まれる」「`~/.profile` が最優先」といった選択肢はいずれも誤りです。

ログアウト時に読まれる対のファイルが `~/.bash_logout` です。**ログインシェルを終了するときにだけ**読み込まれ、画面のクリア(`clear`)や一時ファイルの削除といった後片付けを書きます。「ログイン時に読まれる」「非ログインシェルの終了時にも読まれる」はどちらも誤りです。

非ログインの対話的シェルとして起動した場合

端末アプリを開いたときのように、非ログインで対話的なシェルが起動した場合は `~/.bashrc` が読まれます。全ユーザー共通の対になるファイルは `/etc/bash.bashrc` です。

⚠️ このファイル名はディストリビューションで異なります。**Debian / Ubuntu 系が** `/etc/bash.bashrc` **、Red Hat 系は** `/etc/bashrc` です。試験の出題文言は `/etc/bash.bashrc` を採ります。

一覧で押さえる

ファイル	適用範囲	読み込まれるタイミング
`/etc/profile`	全ユーザー	ログインシェルの起動時(**最初**)
`/etc/profile.d/*.sh`	全ユーザー	`/etc/profile` の中からまとめて読まれる
`~/.bash_profile`	個人	ログインシェルの起動時(3つのうち**最初に見つかった1つだけ**)
`~/.bash_login`	個人	同上(`~/.bash_profile` が無いときに探される)
`~/.profile`	個人	同上(上2つが無いときに探される)
`/etc/bash.bashrc`	全ユーザー	非ログインの対話的シェルの起動時
`~/.bashrc`	個人	非ログインの対話的シェルの起動時
`~/.bash_logout`	個人	**ログインシェルの終了時**

`/etc/profile.d/` —— 全体設定を「足す」正しい場所

`/etc/profile` は、その中から **`/etc/profile.d/` 配下の `.sh` で終わるファイルをまとめて読み込みます**。全ユーザー共通の環境変数やエイリアスを足したいときは、`/etc/profile` 本体を書き換えるのではなく、**`/etc/profile.d/` に `.sh` ファイルを1枚置く**のが作法です。パッケージ更新で本体が上書きされても設定が残る、という利点があります。

まぎらわしい置き場所を切り分けておきます。

場所	何をする場所か	よくある取り違え
`/etc/profile.d/`	全ユーザー共通の設定を**足す**	ここに置くのは `.sh` で終わるファイル。拡張子が違うと読まれない
`/etc/skel/`	**新規ユーザー作成時にホームへ複写される雛形**	既存ユーザーには一切影響しない
`/etc/environment`	PAM が読む `変数=値` の並び	**シェルの構文が書けない**（`export` もエイリアスも変数の参照も不可）

なぜ `~/.profile` から `~/.bashrc` を呼ぶのか

多くの環境では、ログイン用の設定ファイルの中に次の 1 行が入っています。

```
if [ -f ~/.bashrc ]; then . ~/.bashrc; fi
```

「`~/.bashrc` が存在したら読み込む」という意味です。目的は、**ログインシェルでも `~/.bashrc` の内容(エイリアスなど)を有効にすること**です。本来ログインシェルは `~/.bashrc` を読みませんが、この呼び出しを入れておけば、ログイン直後の画面でも端末を開き直したときでも同じ操作感になります。「設定を 1 か所(`~/.bashrc`)にまとめるため」という理由がそのまま設問になります。

書く場所の使い分け ―― 「ログイン時」か「新しいシェルを起動するたび」か

到達目標の「ログイン時、あるいは新しいシェルを起動したときに環境変数(たとえば検索パス)を設定する」がここに当たります。

- **ログイン時に 1 回だけ効かせたい**(環境変数・`umask`・1 回きりの初期化)
  → `~/.bash_profile`(全体なら `/etc/profile`)

- **新しいシェルを起動するたびに効かせたい**(エイリアス・シェル関数・プロンプト) → `~/.bashrc`(全体なら `/etc/bash.bashrc`)

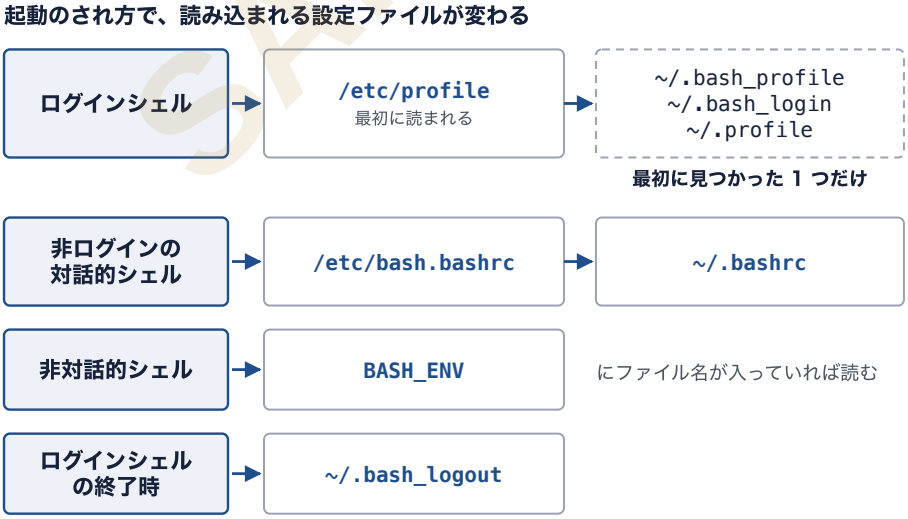
エイリアスと関数は**子プロセスへ引き継がれない**ので、`~/.bash_profile` に書くと端末を開き直したときに消えます。だからエイリアスは `~/.bashrc` に書く、と対で覚えてください。

非対話的シェルの場合

`bash script.sh` のような非対話的シェルは、上のどれも読みません。代わりに環境変数 `BASH_ENV` にファイル名が入っていれば、それを読み込みます。「cron から実行したらエイリアスが効かない」の理由もこれです。

起動時の読み込みを止めるオプションも押さえます。

- `bash --norc` —— `~/.bashrc` を読まない
- `bash --noprofile` —— `/etc/profile` と個人プロファイルを読まない



エイリアスと関数は子プロセスへ引き継がれないので、~/.bashrc に書く
全ユーザー共通の設定を足すなら /etc/profile.d/ に .sh ファイルを 1 枚置く

図 1-2　起動のされ方で、読み込まれる設定ファイルが変わる

- `bash -l` / `bash --login` —— **ログインシェルとして**起動する

1-1-3 シェル変数と環境変数 —— `set` / `env` / `export` / `unset` [105.1]

変数は値に名前を付けて保存する仕組みです。シェルの変数は 2 種類あります。

- **シェル変数** —— そのシェルの中だけで有効。`NAME=value` と書くだけで作れる
- **環境変数** —— そこから起動される**子プロセスにも引き継がれる**。`export NAME=value` のように `export` を付けると環境変数になる

⚠️ 代入は **イコールの前後に空白を入れてはいけません**。`NAME = value` と書くと、`NAME` というコマンドに `=` と `value` を渡した扱いになり「コマンドが見つかりません」となります。記述入力で狙われる書式です。

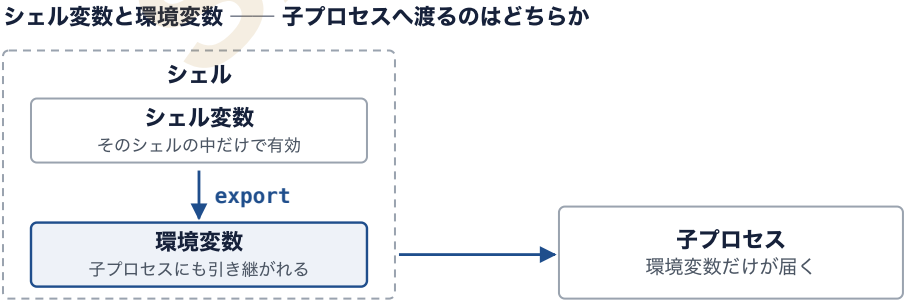
家に例えるなら、シェル変数は「その部屋の中のメモ」、環境変数は「子どもに持たせる手紙」です。子プロセスに渡したいなら `export` が要ります。

値の参照は `$NAME` または `${NAME}` です。未設定のときに代わりの値を使いたければ `${NAME:-代替値}` を書きます。`echo ${USERNAME:-guest}` は `USERNAME` が未設定なら `guest` と表示しますが、**`USERNAME` 自体の値は書き換えません**(書き換えたいなら `${NAME:=代替値}`)。

4 つのコマンドの守備範囲

コマンド	何を表示・操作するか	覚え方
`env`	**環境変数だけ**を一覧表示。環境を変えてコマンドを実行することもできる	環境(environment)専門
`set`	**シェル変数・環境変数・シェル関数のすべて**を一覧表示。シェルの動作オプションも設定する	全部見える
`export`	シェル変数を**環境変数に格上げ**する	子へ渡す
`unset`	変数や関数を**削除**する	消す

「環境変数だけを見たい」なら `env`（または `printenv`）、「関数も含めて全部見たい」なら `set`。この段差がそのまま設問になります。**`set` は変数を作るコマンドではありません** —— 引数なしの `set` は一覧表示、`set -e` などはシェルの動作オプションの設定です。



env —— 環境変数だけを一覧表示(printenv も同じ)
set —— シェル変数・環境変数・シェル関数のすべてを一覧表示
export —— シェル変数を環境変数に格上げする / unset —— 変数や関数を削除する
NAME=value の = の前後に空白を入れてはいけない

図 1-3　シェル変数と環境変数 —— 子プロセスへ渡るのはどちらか

`env` —— 環境変数の表示と、環境を変えた実行

書式　`env [オプション] [変数名=値 ...] [コマンド [引数 ...]]`

オプション	長形	意味
`-i`	`--ignore-environment`	**既存の環境変数をすべて無視**し、空の環境から始める
`-u 名`	`--unset=名`	指定した変数を環境から取り除いて実行する
`-0`	`--null`	出力の区切りを改行ではなく NUL 文字にする
`-C 位置`	`--chdir=位置`	指定ディレクトリへ移動してから実行する

例　`env`

例　`env | grep PATH`

例　`env LANG=C date`

例　`env -i bash --norc`

例　`env -u LANG locale`

`env LANG=C date` は「この 1 回だけ `LANG` を `C` にして `date` を実行する」という意味で、**元のシェルの `LANG` は変わりません。**一時的に環境を変える定番の書き方です。`env` に似た `printenv` は表示専用で、`printenv PATH` のように特定の変数だけを出せます。

なお、スクリプトのシバンで `#!/usr/bin/env bash` と書くのも同じコマンドです。**検索パスから `bash` を探して起動する**ため、`bash` の置き場所が環境によって違っても動く、という利点があります。

`export` —— 環境変数に格上げする

書式　`export [-fnp] [名前[=値] ...]`

オプション	意味
`-p`	現在 export されている変数の一覧を、再利用できる形で表示する
`-n`	export 属性を**外す**(変数自体は消さず、シェル変数に戻す)
`-f`	変数ではなく**シェル関数**を export する

例　`export EDITOR=vim`

例　`VAR=hello; export VAR`

例　`export -p`

例　`export -n VAR`

例　`export -f myfunc`

`export NAME=value` と `declare -x NAME=value` は同じ意味です(`-x` の x は export の x)。逆に `local` は関数の中だけの宣言、`readonly` は値の変更を禁止する宣言で、どちらも子プロセスへは引き継ぎません。

⚠️「export したら親のシェルにも反映される」は誤りです。変数が流れるのは**親から子への一方向だけ**で、子プロセスが何をしても親のシェルの変数は

変わりません。子から親へ伝える方法は無い、という前提が 1-1-6 の source 実行の理解につながります。

`set` —— 一覧表示とシェルの動作オプション

書式　`set [--abefnuvxCo] [オプション名] [引数 ...]`

書き方	意味
`set`	シェル変数・環境変数・シェル関数をすべて一覧表示する
`set -e`	コマンドが 1 つでも失敗(終了ステータスが 0 以外)したら直ちに終了する
`set -u`	未定義の変数を参照したらエラーにする
`set -x`	実行するコマンドを**展開後の内容**でトレース表示する
`set -v`	読み込んだ行を**そのまま**表示する
`set -o`	現在のオプションの状態を一覧表示する
`set +x`	`-x` を**解除**する(プラスで解除、が bash の流儀)
`set -- a b c`	位置パラメータを 1 番目から順に `a` `b` `c` へ**設定し直す**

マイナスで有効化、プラスで解除という向きが直感と逆なので、そのまま誤答肢になります。

`unset` —— 変数・関数を消す

書式　`unset [-fv] 名前 ...`

オプション	意味
`-v`	変数を削除する(既定)
`-f`	**シェル関数**を削除する

例　`unset VAR`

例　`unset -f myfunc`

⚠ **`unset VAR` と `VAR=` は別物**です。`unset` は変数そのものを消すので `${VAR:-代替}` の判定では「未設定」、`VAR=` は「空文字が入っている」状態で残ります。また **`unset` でエイリアスは消せません** —— エイリアスを消すのは `unalias` です。ここは 1 対 1 で対応させて覚えてください。

1-1-4 検索パス `PATH` を正しく設定する [105.1]

`PATH` は、コマンド名だけを打ったときにシェルが実行ファイルを探しに行くディレクトリの一覧です。**コロン区切り**で、**左から順に探し、最初に見つかったものを実行**します。到達目標「適切なディレクトリでコマンドの検索パスを設定する」がここに当たります。

追加するときは、**既存の値を必ず残す**のが鉄則です。

- 末尾に追加(既存のコマンドを優先): `export PATH=$PATH:/opt/bin`
- 先頭に追加(新しい場所を優先): `export PATH=/opt/bin:$PATH`

⚠️ `export PATH=/opt/bin` と書くと**それまでの検索パスがすべて消え**、`ls` すら「コマンドが見つかりません」になります。試験では「既存の検索パスをすべて維持したまま `/opt/bin` を加える」という条件付きで出るので、式の中に `$PATH` が入っているかを必ず確かめてください。記述入力でも出ます。

カレントディレクトリは既定では検索パスに含まれません。そのため、カレントディレクトリに実行権限付きの `myscript.sh` があっても `myscript.sh` とだけ打つと見つかりません。`./myscript.sh` のようにパスを明示するのが正解です。これは、細工した `ls` を置いた作業ディレクトリに管理者を誘い込む攻撃を防ぐための意図的な設計で、「カレントディレクトリを検索パスに入れるべきか」という問いには「セキュリティ上入れない」が答えになります。

検索パスの効き方を調べる道具も押さえます。

コマンド	何が分かるか
`which ls`	検索パスの中から見つかった実行ファイルのパス
`type ls`	エイリアス・シェル関数・ビルトイン・外部コマンドの**どれか**まで分かる
`type -a ls`	同名のものを**すべて**、優先順に列挙する
`hash -r`	シェルが覚えているコマンド位置のキャッシュを破棄する

`type` が `which` より強いのは、**エイリアスや関数で上書きされている場合を見抜ける**点です。「`ls` が見慣れない色で出る」の原因調査は `type ls` が正解肢になります。

⚠️ `~/.bashrc` に検索パスの追加行を書いた場合、効くのは**それ以降に起動したシェル**です。すでに開いている端末に効かせたいなら `source ~/.bashrc`

を実行します(1-1-6)。

1-1-5 エイリアスとシェル関数 [105.1]

alias / unalias —— コマンドに別名を付ける

書式 `alias [名前[='コマンド']]`

書式 `unalias [-a] 名前 ...`

書き方	意味
`alias`	定義済みエイリアスを一覧表示する
`alias ll='ls -l'`	エイリアスを定義する(**イコールの前後に空白を入れない**)
`alias ll`	`ll` の定義内容だけを表示する
`unalias ll`	`ll` の定義を削除する
`unalias -a`	**すべての**エイリアスを削除する

定義に空白やオプションを含めるときは**クォートで囲みます**。`alias cp='cp -i'` のようにコマンド名と同じ名前も定義でき、その場合エイリアスが優先されます。**エイリアスを一時的に迂回して本来のコマンドを実行したい**ときは、次のいずれかを使います。

- `\cp a b` —— 先頭にバックスラッシュ
- `command cp a b` —— `command` 経由で外部コマンドを直接呼ぶ
- `/bin/cp a b` —— 絶対パスで呼ぶ

⚠️ **ターミナルで直接定義したエイリアスは、そのシェルを終了すると消えます。**恒久的に使いたければ `~/.bashrc` に書きます。「次回ログイン以降も有効にするには?」と来たら設定ファイルへの記述が答えです。

⚠️ **エイリアスは引数を受け取れません。**`alias mkcd='mkdir $1 && cd $1'` のような書き方は期待どおり動きません(引数は展開後の末尾に付くだけです)。引数を扱いたい・複数行の処理をしたい場合は**シェル関数**を使います。この段差が 105.1 の「よく使うコマンド列を Bash 関数として書く」に当たります。

function —— よく使う手順に名前を付ける

到達目標に「よく使うコマンド列を Bash 関数として書けること」が明記されています。定義の書き方は 3 通りあり、**どれも正しい構文**です。

```
書式  function 名前 [()] { コマンド ...; }
書式  名前() { コマンド ...; }
```

- `function greet { echo Hello; }` —— `function` あり・丸かっこ無し
- `greet() { echo Hello; }` —— `function` 無し・丸かっこあり
- `function greet() { echo Hello; }` —— 両方あり(**これも誤りではありません**)

不可なのは `function` も丸かっこも無い形(`greet { echo Hello; }`)だけです。中かっこの**内側には空白が要り**、最後のコマンドの後に セミコロン が要ります(`{ echo Hello; }`)。

引数はスクリプトと同じく `$1` `$2` … `$#` `$@` で受け取ります。関数の中だけで有効な変数は `local` を付けて宣言します。値を返すのは `return`(0〜255

の終了ステータス)で、文字列を返したいときは `echo` して呼び出し側でコマンド置換を使います。

```
mkcd() { mkdir -p "$1" && cd "$1"; }
backup() { local d=$(date +%F); cp "$1" "$1.$d"; }
```

	エイリアス	シェル関数
引数の扱い	受け取れない	`$1` `$2` … で受け取れる
複数行・分岐・ループ	書けない	書ける
定義するコマンド	`alias`	`function` または `名前()`
一覧表示	`alias`	`declare -f`
削除	`unalias`	`unset -f`
子プロセスへ渡す	渡せない	`export -f` で渡せる
恒久化する場所	`~/.bashrc`	`~/.bashrc`

関数とエイリアスと外部コマンドで同じ名前があるとき、**エイリアス → 関数 → ビルトイン → 外部コマンド**の順で選ばれます。`type -a 名前` で実際の優先順を確認できます。

1-1-6 `.` と `source` —— 変更を現在のシェルに反映する [105.1]

シェルスクリプトを `./setenv.sh` や `bash setenv.sh` で実行すると、**新しい子シェル(サブシェル)が起動してそこで実行され、終わると消えます**。スクリプトの中で `export MYVAR=hello` としても、その変数は子シェルのものなので、**元の**

対話シェルには残りません。1-1-3 で見た「変数は親から子への一方向」がそのまま効いています。

現在のシェルに反映させたいときは **source 実行**を使います。

```
書式　source ファイル名 [引数 ...]
書式　. ファイル名 [引数 ...]
```

- `source ~/.bashrc`
- `. ~/.bashrc` —— **ドットと空白。**上とまったく同じ意味

この 2 つは同義で、**新しいシェルを起動せず、現在のシェルの中で 1 行ずつ実行する**のが特徴です。だから変数もエイリアスも関数もそのまま残ります。

⚠ **`.`（ドット）は POSIX 準拠の元祖、`source` は bash の別名**です。`#!/bin/sh` で書いた移植性重視のスクリプトでは `.` を使います。記述入力では「1 文字で答えよ」の形で `.` が問われることがあります。ドットの後には**空白が必要**で、`.~/.bashrc` は誤りです。

覚え方は「**設定ファイルを書き換えたら source、スクリプトを動かすなら実行**」です。「スクリプト内で定義した変数を、実行後も対話シェルで使いたい」と来たら source 実行、と反射で答えてください。

起動の仕方	どこで動くか	変数は残るか
`./script.sh`	子シェル(実行権限が必要)	残らない
`bash script.sh`	子シェル(実行権限は不要)	残らない
`source script.sh` / `. script.sh`	**現在のシェル**	**残る**

起動の仕方	どこで動くか	変数は残るか
`exec bash script.sh`	現在のシェルを置き換える	シェルごと入れ替わる(1-2-7)

1-1-7 `/etc/skel` —— 新規ユーザー用の雛形ディレクトリ [105.1]

到達目標の「新規ユーザーアカウント用の雛形(skeleton)ディレクトリを保守する」がここです。

`/etc/skel`(skeleton = 骨組み)は、**`useradd` で新しいユーザーを作るときに、その中身がまるごと新しいホームディレクトリへ複写される雛形**です。標準では `.bash_profile`・`.bashrc`・`.bash_logout` が置かれています。全社共通のエイリアスやプロキシ設定を、今後作るユーザー全員に持たせたいときは、ここへファイルを置きます。

⚠️ **`/etc/skel` を変更しても、すでに存在するユーザーには一切影響しません。**複写は**アカウント作成時の 1 回だけ**です。「全ユーザーに今すぐ効かせたい」なら `/etc/profile.d/` に置く、「これから作るユーザーに持たせたい」なら `/etc/skel` に置く —— この対比が設問になります。

複写が起きるのは **ホームディレクトリが作られるとき**なので、`useradd` に **`-m`(`--create-home`)**が要ります(既定で `-m` 相当になるディストリビューションもあります)。別の雛形を使いたいときは **`-k ディレクトリ`(`--skel`)**で指定します。

```
例  useradd -m taro
例  useradd -m -k /etc/skel.dev hanako
```

雛形の場所の既定値は `/etc/default/useradd` の `SKEL=` 行に書かれており、`useradd -D` で現在の既定値を確認できます。ドットで始まるファイル(隠しファイル)も複写対象で、複写後の**所有者は新しいユーザーに変更**されます。

1-2 簡単なスクリプトをカスタマイズする [105.2]

到達目標は「既存のスクリプトをカスタマイズできること、簡単な Bash スクリプトを新しく書けること」です。本格的なプログラミングは要りません。問われるのは **標準的な sh の構文(ループとテスト)を正確に書けるか、コマンドの成否をどう受け取って分岐するか**、そして **スクリプトの置き場所と権限を管理できるか** の 3 点です。

1-2-1 スクリプトの骨格 —— シバン(`#!`)で解釈するシェルを決める [105.2]

シェルスクリプトは、実行したいコマンドを上から順に並べたテキストファイルです。**1 行目の先頭に `#!` を書き、続けてインタプリタの絶対パスを書きます**。この行を**シバン**(shebang)と呼びます。

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

到達目標の「シバン行によってスクリプトのインタプリタを正しく選択する」がここです。押さえどころを 4 つ挙げます。

- **必ず 1 行目**であること。2 行目以降に書いてもただのコメントとして無視されます
- **`#` と `!` の間、`!` とパスの間に空白を入れない**(`#! /bin/bash` は許容する実装もありますが、試験では `#!/bin/bash` を正とします)

- **絶対パスで書く。**`#!/bin/bash`・`#!/bin/sh`・`#!/usr/bin/perl`・`#!/usr/bin/python3` など
- 検索パス経由で解決したいときは **`#!/usr/bin/env bash`** と書く

シバンが効くのは **`./script.sh` のように「実行」したとき**です。`bash script.sh` と明示的にインタプリタを指定して起動した場合、シバンは無視され、指定した `bash` で解釈されます。ここが誤答肢の作りどころです。

呼び出し方	実行権限	シバンは効くか
`./script.sh`	**必要**	効く
`bash script.sh`	不要	効かない(`bash` で解釈)
`sh script.sh`	不要	効かない(`sh` で解釈)
`source script.sh` / `. script.sh`	不要	効かない(現在のシェルで解釈)

⚠️ **`#!/bin/sh` と `#!/bin/bash` は同じではありません。**多くのディストリビューションで `/bin/sh` は `dash` などへのシンボリックリンクになっており、`[[ ]]`・配列・`function` キーワードといった bash 拡張が使えません。「bash 特有の書き方を使っているのに `#!/bin/sh` と書いたので動かない」というのが典型的な故障パターンです。

`#` から行末まではコメントです(1 行目のシバンだけが例外的な意味を持ちます)。

1-2-2 スクリプトの置き場所・所有者・実行権限・SUID [105.2]

到達目標に「スクリプトの場所・所有権・実行権限・suid 権限を管理する」と明記されています。ここは 4 つに分けて押さえます。

置き場所(location)

置き場所	用途
`/usr/local/bin`	**管理者が用意した、全ユーザー向けの**スクリプト。パッケージ管理外の自作物はここが定位置
`/usr/local/sbin`	全ユーザー向けではなく、**管理者用**の自作スクリプト
`~/bin`	**その利用者専用**のスクリプト(検索パスに自動で加えるディストリビューションが多い)
`/usr/bin`・`/sbin`	ディストリビューションのパッケージが管理する領域。**自作物を置かない**

置いただけでは名前だけで起動できません。**その場所が検索パスに入っているか**を確認します(1-1-4)。入っていなければ `export PATH=$PATH:/usr/local/bin` のように加えるか、絶対パスで呼びます。

実行権限(execution)

作っただけでは実行できません。**実行権限**が要ります。

書式　`chmod [オプション] モード ファイル ...`

書き方	意味
`chmod +x greet.sh`	全員に実行権限を付ける
`chmod u+x script.sh`	所有者だけに付ける
`chmod ug+x deploy.sh`	所有者とグループに付け、その他には付けない

書き方	意味
`chmod o-x script.sh`	その他のユーザーから実行権限を外す
`chmod 750 script.sh`	数値指定。`rwxr-x---` になる
`chmod -R +x scripts/`	配下すべてに再帰的に適用する

数値は **4 = 読み、2 = 書き、1 = 実行**の足し算です。`750` は「所有者 = 読み書き実行(7)、グループ = 読み+実行(5)、その他 = 権限なし(0)」を意味します。

⚠️ **読み取り権限が無いスクリプトは、実行権限があっても `./script.sh` では動きません。**シェルが中身を読んで解釈する必要があるからです(コンパイル済みバイナリは読み取り権限が無くても実行できる、という対比で出ます)。

所有権(ownership)

書式　`chown [オプション] 所有者[:グループ] ファイル ...`

書き方	意味
`chown user1 backup.sh`	所有者だけを変更する
`chown user1:staff backup.sh`	所有者とグループを同時に変更する
`chown :staff backup.sh`	グループだけを変更する(`chgrp staff backup.sh` と同じ)
`chown -R webuser:webgroup /srv/data`	配下すべてを**再帰的に**変更する

`-R` が再帰、コロンでグループを併記、この 2 点が問われます。所有者を変更できるのは原則 root だけです。

SUID(suid-rights)

SUID(Set User ID)は、**実行したユーザーではなく、そのファイルの所有者の権限で動かす**特殊なパーミッションです。`chmod u+s file` または `chmod 4755 file` で設定し、`ls -l` では所有者の実行権限の位置が **`s`** になります(`-rwsr-xr-x`)。

⚠️ ここが 105.2 の最重要のひっかけです。**Linux のカーネルは、シェルスクリプトに付けられた SUID を無視します。**セキュリティ上の理由(スクリプト実行中に中身を差し替えられる競合の危険があるため)で、意図的に無効化されています。`chmod u+s script.sh` はエラーにならず設定はできますが、**実行しても所有者の権限にはなりません。**

状況	結果
SUID を付けたコンパイル済みバイナリ(`passwd` など)	所有者(root)の権限で動く
SUID を付けたシェルスクリプト	**SUID は無視され、実行者の権限で動く**

一般ユーザーに管理者権限の作業をさせたいときの正解は、SUID ではなく **`sudo` の設定(`/etc/sudoers`)で該当スクリプトの実行だけを許可する**ことです。「スクリプトに SUID を付ければよい」という選択肢は常に誤りだと覚えてください。

関連する特殊パーミッションも対比で押さえます。**SGID**(`chmod g+s`・数値 2000 の位)はディレクトリに付けると「その中に作られたファイルのグループが、ディレクトリのグループを継承する」という効果を持ちます。**スティッキービット**(`chmod +t`・数値 1000 の位)は `/tmp` のように「自分が作ったファイルは自分しか消せない」を実現します。

誰が実行しても同じに動かす

どのユーザーが実行しても同じように動くスクリプトにしたいなら、中で使うコマンドを絶対パスで書くか、先頭で検索パスを明示的に設定します。

```
#!/bin/bash
PATH=/usr/local/bin:/usr/bin:/bin
export PATH
```

呼び出した人の環境変数に結果を左右されないための定石で、cron から実行するスクリプトでは特に重要です(cron の環境は対話シェルとは別物で、検索パスがごく短いためです)。

1-2-3 入力を受け取る —— 位置パラメータと `read` [105.2]

位置パラメータ

スクリプトに渡された引数は、**位置パラメータ**という特別な変数で受け取ります。

変数	意味
`$0`	スクリプト自身の呼び出しパス(**引数ではない**)
`$1` `$2` `$3` …	1 番目、2 番目、3 番目…の引数
`$#`	引数の**個数**(`$0` は数に入らない)
`$*` / `$@`	引数**すべて**
`$?`	直前のコマンドの**終了ステータス**
`$$`	実行中のシェル自身のプロセス ID

変数	意味
`$!`	直前にバックグラウンドで起動したプロセスの ID

`$0` は「1 番目の引数」ではなく**スクリプト名**です。頻出のひっかけです。

`$*` と `$@` は、**ダブルクォートで囲んだときだけ**動作が変わります。`"$*"` は全引数を空白でつないだ**1 つの文字列**、`"$@"` は**引数ごとに分かれた複数の値**になります。したがって引数が 3 つあるとき、`for w in "$@"` のループは 3 回まわり、`for w in "$*"` のループは 1 回しかまわりません。

shift は位置パラメータを 1 つ左へずらすコマンドです。実行すると、それまでの `$2` が新しい `$1` になり、`$#` は 1 減ります。`echo $1; shift; echo $1; echo $#` を `./process.sh one two three` で実行すると `one` → `two` → `2` と表示されます。

read —— 標準入力から 1 行読み込む

書式　`read [オプション] [変数名 ...]`

オプション	意味
`-p 文字列`	入力を促す**プロンプト**を表示してから読み込む
`-s`	入力した文字を**画面に表示しない**(パスワード入力向け)
`-n 数`	指定した文字数を読んだ時点で、改行を待たずに終了する
`-t 秒`	指定秒でタイムアウトする(時間切れなら終了ステータスが 0 以外)

オプション	意味
`-r`	バックスラッシュをエスケープとして解釈しない(**そのまま読む**)
`-a 配列名`	読み込んだ語を配列に格納する

例 `read name`

例 `read -p "名前を入力してください: " name`

例 `read -s -p "パスワード: " pass`

例 `read -r line`

例 `while read line; do echo "$line"; done < list.txt`

押さえどころは 3 つです。

- **変数名の前にドル記号は付けません。**`read name` であって `read $name` ではありません(記述入力で狙われます)
- 変数名を**複数書くと、入力を空白で区切って順に入れます。**余った分は最後の変数にまとめて入ります。変数名を省略すると `REPLY` に入ります
- ファイルの各行を処理する定型句は `while read 変数; do … done < ファイル` です。ファイルの終わり(EOF)に達すると `read` は 0 以外を返し、ループが終わります

`-r` を付けた `read -r` は、バックスラッシュを含む行をそのまま読みたいときの標準的な書き方です。パスを含むファイルを 1 行ずつ処理するときは `while read -r line` と書きます。

1-2-4 終了ステータスと `test` —— 成否を受け取る [105.2]

到達目標「コマンドが返す成功・失敗、あるいはそれ以外の情報を、戻り値で判定する」がここです。

終了ステータス

コマンドは終了時に **終了ステータス**(exit status / 戻り値)という数値を返します。

- **0 = 成功、0 以外 = 失敗**。0 が成功であることに注意してください(「1 が成功」は誤り)
- 直前のコマンドの終了ステータスを見る: `echo $?`
- スクリプト自身を終了ステータス付きで終える: `exit 1`(異常)、`exit 0`(正常)

`exit 2` を実行すると、その時点でスクリプトは終了し、呼び出し元に 2 が返ります。「その行以降は実行されない」がポイントです。`exit` に数値を付けない場合は、**直前のコマンドの終了ステータス**がそのまま返ります。

終了ステータスは 0〜255 の範囲です。コマンドによっては「1 = 一致なし、2 = ファイルが無い」のように**失敗の種類を数値で区別**します。`grep` がその代表で、`grep -q 語 file` は**一致があれば 0、無ければ 1、ファイルが読めなければ 2** を返します。「成功・失敗以外の情報」とはこのことです。

`set -e` を書いておくと、それ以降のコマンドが 1 つでも失敗した時点でスクリプトが直ちに終了します。エラーを握りつぶさないための定石です。

`test` と `[ ]`

条件を判定するのは `test` コマンド、または**まったく同じ意味**の `[ ]` です。

```
書式　test 式
書式　[ 式 ]
```

⚠️ **角かっこの内側には必ず空白が要ります。**`[ -f file ]` であって `[-f file]` ではありません。`[` は記号ではなく**コマンド名**なので、後ろに空白が必要なのです。閉じる `]` の前にも空白が要ります。記述入力で最も落とされる書式です。

ファイルに関する判定。

演算子	真になる条件
`-e ファイル`	種類を問わず**存在する**
`-f ファイル`	**通常ファイル**として存在する
`-d ファイル`	**ディレクトリ**として存在する
`-L ファイル`	シンボリックリンクである
`-s ファイル`	存在し、**サイズが 0 より大きい**
`-r` / `-w` / `-x`	読み取り / 書き込み / 実行の権限がある
`-u` / `-g`	SUID / SGID が設定されている
`ファイル1 -nt ファイル2`	ファイル1 のほうが新しい(newer than)
`ファイル1 -ot ファイル2`	ファイル1 のほうが古い(older than)

数値と文字列の比較。**ここを取り違えるのが定番の誤答**です。

比べたいもの	等しい	等しくない	より小さい	以下	より大きい	以上
数値	`-eq`	`-ne`	`-lt`	`-le`	`-gt`	`-ge`
文字列	`=`	`!=`	—	—	—	—

数値の比較に `<` や `>` を使わないのが sh の流儀です。`[ $a > $b ]` と書くと比較ではなく**リダイレクトと解釈され、`b` という名前のファイルが作られてしまいます**。この結果まで含めて設問になります。

文字列の判定には `-z 文字列` (**長さが 0 なら真**)と `-n 文字列` (**長さが 0 より大きければ真**)もあります。`-z` の z は zero、`-n` の n は nonzero と覚えます。

複数の条件をつなぐ書き方も押さえます。

- `[ 条件A -a 条件B ]` —— かつ(AND)
- `[ 条件A -o 条件B ]` —— または(OR)
- `[ ! 条件 ]` —— 否定(NOT)
- `[ 条件A ] && [ 条件B ]` —— 現在推奨される書き方

⚠ **変数はダブルクォートで囲む**のが安全な書き方です。`[ -z $VAR ]` は `VAR` が未設定だと引数が消えて構文エラーになりますが、`[ -z "$VAR" ]` なら空文字として正しく判定されます。

`[[ 式 ]]` は bash の拡張で、パターンマッチや `&&` `||` を内側に書けます。ただし **POSIX の `sh` では使えない**ので、`#!/bin/sh` のスクリプトでは `[ ]` を使います。

1-2-5 コマンドの連結 —— `;` と `&&` と `||` [105.2]

到達目標「連結したコマンドを実行する」がここです。複数のコマンドを 1 行でつなぐ記号は 3 つあり、**直前のコマンドの終了ステータスをどう扱うか**が違います。択一でも記述入力でも繰り返し出ます。

記号	意味	直前のコマンドが失敗したら
`;`	成否に関係なく順番に実行する	それでも次を実行する
`&&`	直前が**成功(終了ステータス 0)**したときだけ次を実行	次は実行されない
`||`	直前が**失敗(0 以外)**したときだけ次を実行	そのときだけ次を実行する

例として `mkdir /tmp/work && cd /tmp/work || echo 処理を中断しました` を読み解きます。`mkdir` が失敗すると `&&` の右側(`cd`)は実行されず、`||` の右側が拾って「処理を中断しました」が表示されます。逆に `mkdir` が成功すれば `cd` が実行され、その `cd` も成功するので `||` の右側は実行されません。

`grep -q エラー app.log || echo 見つかりません` も同じ読み方です。`grep -q` は**一致があれば成功、無ければ失敗**を返すので、これは「見つからなかったときだけメッセージを出す」という定型句になります。

`&&` と `||` は左から順に評価されるので、`A && B || C` は「A が成功したら B、**A か B のどちらかが失敗したら** C」という意味になります。「A が失敗したときだけ C」ではない点に注意してください。厳密に分岐させたいときは `if` を使います。

関連する記号も対比で押さえます。

記号	意味
`&`	コマンドを**バックグラウンド**で実行する(次の行へすぐ進む)
`|`	左のコマンドの標準出力を、右のコマンドの標準入力へ渡す(**パイプ**)
`;`	順に実行する
`&&` / `||`	終了ステータスによる分岐

`&` (1 つ)はバックグラウンド実行、`&&` (2 つ)は成功時の連結です。1 文字違いでまったく意味が違うので、そのまま誤答肢になります。

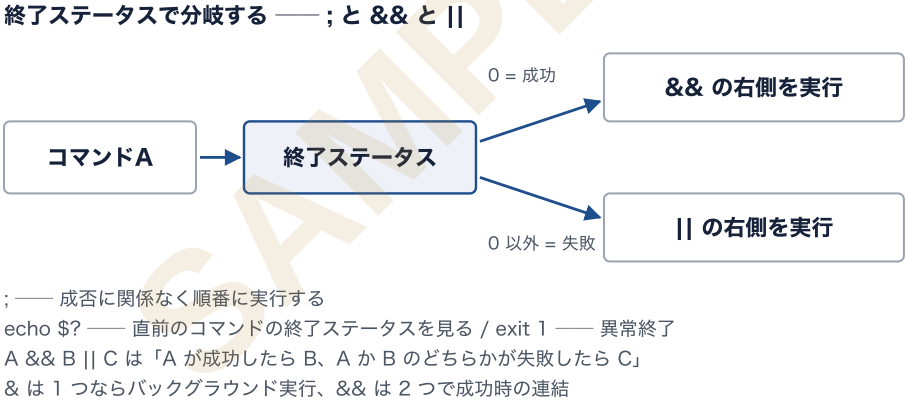


図 1-4　終了ステータスで分岐する ―― ; と && と ||

1-2-6 コマンド置換 ―― 実行結果を値として埋め込む [105.2]

到達目標「コマンド置換を使う」がここです。**コマンド置換**は、コマンドの実行結果(標準出力)をその場の文字列として埋め込む書き方です。

- `$(コマンド)` ―― 現在の書き方

- `` `コマンド` `` —— バッククォートで囲む古い書き方(意味は同じ)

```
例  echo "今日は $(date +%F) です"
例  NOW=$(date +%F)
例  COUNT=$(ls -1 /var/log | wc -l)
例  for f in $(cat list.txt); do echo "$f"; done
```

`$( )` は入れ子にできる点がバッククォートに対する利点です(`$(dirname $(which bash))` のように書けます)。バッククォートで入れ子にするとエスケープが必要になり読みにくいため、現在は `$( )` が推奨されます。

⚠️ 混同しやすい 3 つの丸かっこ・波かっこを切り分けます。

書き方	意味
`$(コマンド)`	**コマンド置換**。コマンドの出力に置き換わる
`$((式))`	**算術演算**。`$((2 + 3))` は `5` になる
`${変数}`	**変数展開**。変数の値に置き換わる
`(コマンド)`	**サブシェル**で実行する(中の `cd` や変数は外に影響しない)

`$(( ))` と `$( )` の取り違えは頻出です。**かっこが 2 重なら計算、1 重ならコマンド**と覚えてください。

コマンド置換の結果は**末尾の改行が取り除かれ**、クォートで囲まなければ空白で単語分割されます。ファイル名に空白が含まれる可能性がある場面では `"$(コマンド)"` とダブルクォートで囲みます。

1-2-7 条件分岐 —— `if` と `case` [105.2]

到達目標「標準的な sh の構文(ループ、テスト)を使う」の前半です。

`if` 文

書式　`if 条件; then 処理; [elif 条件; then 処理;] [else 処理;] fi`

```
if [ -f config.conf ]; then
  echo found
elif [ -d config.d ]; then
  echo directory
else
  echo missing
fi
```

1 行で書くときは `if [ -f config.conf ]; then echo found; fi` です。**`then` の前にセミコロンが要る**こと、**終わりが `fi`** (if の逆さ読み)であることが問われます。`endif` や `end if` という選択肢は誤りです。

⚠ `if` が見ているのは**コマンドの終了ステータス**であって、真偽値ではありません。だから `if [ … ]` だけでなく **`if grep -q 語 file; then …` のようにコマンドを直接書けます**。`[ ]` は「判定してくれるコマンド」の 1 つにすぎない、という理解が誤答肢を切る鍵になります。

`case` 文

値によって処理を振り分ける構文です。

書式　case 値 in パターン) 処理;; パターン) 処理;; *) 処理;; esac

```
case "$1" in
  start) echo 起動します;;
  stop)  echo 停止します;;
  restart|reload) echo 再読み込みします;;
  *)     echo "使い方: $0 {start|stop|restart}";;
esac
```

押さえどころは 4 つです。

- 終わりは **esac**（case の逆さ読み）
- 各分岐の終わりは **;;**（セミコロン 2 つ）。1 つでは構文エラーになります
- パターンの後ろは**閉じかっこだけ**（`start)`）。前に開きかっこは要りません（付けても構いません）
- 縦棒で複数のパターンを並べられます（`restart|reload)`）。**最後の `*)` が「どれにも当てはまらない場合」**を受け持ちます

パターンには **`*` `?` `[]` のワイルドカードが使えます**。`[Yy]*)` と書けば `Y` または `y` で始まる入力をまとめて受け取れます。正規表現ではなくファイル名と同じパターンである点に注意してください。

`if` と `case` の使い分けは、**1 つの値を複数の候補と突き合わせるなら `case`、条件が値の比較にとどまらない（ファイルの存在・権限など）なら `if`** です。

1-2-8 ループ —— `for` / `while` / `until` と `seq` [105.2]

`for` 文

決まった並びを順に処理します。終わりは `done` です。

```
書式  for 変数 in 並び; do 処理; done
```

```
for fruit in apple banana cherry; do echo $fruit; done
for f in *.txt; do echo "$f"; done
for user in $(cat users.txt); do echo "$user"; done
```

連続する数値を回す書き方は 3 通りあり、**どれも正解**になります。

- `for i in 1 2 3 4 5; do echo $i; done` —— 直接並べる
- `for i in {1..5}; do echo $i; done` —— bash の**ブレース展開**。`{1..5}` は `1 2 3 4 5` に展開される(**ピリオド 2 つ**・空白を入れない)
- `for i in $(seq 1 5); do echo $i; done` —— **コマンド置換**で `seq` の出力を並びとして使う

⚠ **`in 1..5`(ブレース無し)、`in [1-5]`、`in 1,2,3,4,5` はいずれも展開されません。**その文字列 1 個をそのまま代入して 1 回しか回らないので、ここが誤答の作りどころです。`[1-5]` はファイル名のパターンとしては働きますが、該当するファイルが無ければ文字列のまま残ります。

`in 並び` を省略した `for 変数; do … done` は、**位置パラメータ(`"$@"`)を順に回す**という意味になります。C 言語風の `for ((i=0; i<5; i++))` も bash では使えますが、これは POSIX の `sh` にはない拡張です。

`seq` —— 連続した数値を出力する

書式　`seq [オプション] [開始 [増分]] 終了`

書き方	出力
`seq 5`	1 から 5 まで(開始を省略すると 1 から)
`seq 2 6`	2 3 4 5 6
`seq 1 2 9`	1 3 5 7 9(**真ん中が増分**)
`seq 5 -1 1`	5 4 3 2 1(増分を負にすると降順)

オプション	長形	意味
`-s 文字列`	`--separator`	区切り文字を指定する(既定は改行)
`-w`	`--equal-width`	桁数を **0 埋めでそろえる**(`seq -w 8 10` は `08 09 10`)
`-f 書式`	`--format`	printf 形式で書式を指定する

⚠️ 3 つ引数を書いたときの並びは **`開始 増分 終了`** です。「開始 終了 増分」と覚えていると `seq 1 9 2` を `1 3 5 7 9` と誤読します。実際には `1` と `9`(増分 9)しか出ません。ここは設問で狙われます。

`seq -s , 1 5` は `1,2,3,4,5` を 1 行で出力します。`for` と組み合わせる用途のほか、`seq 1 100 | xargs …` のように連番を作る道具として使います。

while 文と until 文

書式　while 条件; do 処理; done

書式　until 条件; do 処理; done

while は条件が真である間繰り返し、**until は条件が真になるまで**繰り返します。判定の向きが正反対です。

```
count=1
while [ $count -le 3 ]; do
  echo $count
  count=$((count + 1))
done
```

```
until ping -c1 -W1 192.168.1.1 > /dev/null 2>&1; do
  echo 応答がありません
  sleep 5
done
```

ファイルを 1 行ずつ処理する定型句は 1-2-3 で見た形です。

```
while read -r line; do echo "$line"; done < list.txt
```

ループの中で使う制御は 2 つです。**break** はループを抜け、**continue** はその回だけ飛ばして次の繰り返しに進みます。`break 2` のように数値を付けると、入れ子になったループを指定した段数まとめて抜けます。`while true; do … done` のような無限ループは、中の `break` で抜けるのが前提です。

⚠️ `while` の終わりも **`done`** です。`for` `while` `until` はすべて `do` で始まり `done` で終わり、`if` だけが `then` … `fi`、`case` だけが `in` … `esac` である、と 3 系統で整理して覚えてください。

1-2-9 `exec` —— 現在のプロセスを置き換える・出力先を変える [105.2]

```
書式  exec [コマンド [引数 ...]]
書式  exec [リダイレクト]
```

`exec` はコマンドを実行するときに、新しいプロセスを作らず、現在のシェルのプロセスをそのコマンドで置き換えます。 置き換わるので、そのコマンドが終わってもシェルには戻らず、プロセスごと終了します。

書き方	起きること
`bash other.sh`	子プロセスが作られ、終わると元のシェルに戻る
`exec bash other.sh`	**現在のシェルが置き換わる。**終わると元のシェルには戻らない
`exec /bin/false`	置き換わったコマンドが終了した時点でシェルも終わる(端末が閉じる)

用途は 2 つです。**1 つは、ラッパースクリプトの最後で本来のプログラムに処理を明け渡すとき** —— プロセスを 1 つ節約でき、シグナルもそのまま届きます。

もう 1 つがリダイレクトの固定です。コマンドを書かずにリダイレクトだけを指定すると、**それ以降のスクリプト全体**の入出力先が変わります。

```
exec > /var/log/mybatch.log 2>&1
echo ここから先の出力はすべてログファイルへ行きます
```

```
exec 3< input.txt
read -r line <&3
exec 3<&-
```

上は「以降の標準出力と標準エラー出力をログファイルへ向ける」、下は「ファイルディスクリプタ 3 番を開いて読み、最後に閉じる」書き方です。`2>&1` が「標準エラー出力を標準出力と同じ先へ」を意味する点も合わせて押さえます。

⚠ **対話シェルで `exec` に失敗すると、そのシェルは終了します**(存在しないコマンドを `exec` すると端末が閉じる)。「`exec` は子プロセスを作る」という選択肢は誤りで、**作らないのが `exec` の定義**です。

1-2-10 スーパーユーザーへの条件付きメール送信 [105.2]

到達目標に「スーパーユーザーへの条件付きメール送信を行う」と明記されています。バッチ処理の中で異常を検知したときだけ管理者へ通知する、という定型です。

```
書式  mail [-s 件名] 宛先
```

オプション	意味
`-s 件名`	件名(Subject)を指定する
`-c アドレス`	Cc に加える

オプション	意味
`-b アドレス`	Bcc に加える
`-a ファイル`	ファイルを添付する(実装による)

例　`echo "ディスク使用率が閾値を超えました" | mail -s "disk alert" root`

例　`mail -s "backup log" root < /var/log/backup.log`

本文は**標準入力から**渡します。パイプで流す形と、リダイレクトでファイルの中身を流す形の 2 つが定番です。宛先の `root` がスーパーユーザーを指します(`mail -s "件名" root` の綴りをそのまま打てるようにしてください)。コマンド名は環境によって `mail` のほか `mailx`・`sendmail` が使われます。

「条件付き」の部分は、1-2-4・1-2-5 で見た終了ステータスの分岐と組み合わせます。

```
if ! /usr/local/bin/backup.sh; then
  echo "バックアップに失敗しました" | mail -s "backup failed" root
fi
```

```
/usr/local/bin/backup.sh || echo "バックアップに失敗しました" | mail -s "backup failed" root
```

```
grep -q "ERROR" /var/log/app.log && mail -s "error found" root < /var/log/app.log
```

3 つとも「**うまくいかなかったとき(あるいは特定の状況のとき)だけ** root へ通知する」という同じ意図の書き方です。`||` は失敗時、`&&` は成功時、という向きを取り違えないでください。

⚠️ cron が実行したコマンドが**標準出力や標準エラー出力に何かを書くと、cron はその内容を実行ユーザー宛にメール送信します**。逆に言えば、通知したくない出力は `> /dev/null 2>&1` で捨て、**通知したい異常のときだけ出力する**のがバッチ設計の定石です。crontab の中で `MAILTO=root` と書けば、宛先を root に固定できます。`MAILTO=""` と空にすればメールは送られません。

1-2-11 スクリプトのデバッグ —— `-x` と `-v` [105.2]

スクリプトが期待どおり動かないときの調査方法も出題されます。2 つのオプションの違いが核心です。

書き方	表示されるもの
`bash -x script.sh`	実行される各コマンドを、**変数を展開した後の実際の内容**で表示する
`bash -v script.sh`	シェルが**読み込んだ行をそのまま**(展開前の記述どおり)表示する
`bash -n script.sh`	**実行せずに構文チェックだけ**を行う

`-x` は「実際に何が実行されたか」、`-v` は「何と書いてあるか」、`-n` は「実行せず文法だけ見る」です。変数の中身が想定と違うことを疑うなら `-x`、構文そのものを追いたいなら `-v`、本番前の点検なら `-n` を選びます。

スクリプトの一部だけを追いたいときは、中に `set -x` と `set +x` を書いて挟みます(1-1-3)。`-x` の出力は各行の先頭に `+` が付き、標準エラー出力へ出ます。

第1章の試験ポイント [105.1/105.2]

- ログインシェルは `/etc/profile` → `~/.bash_profile` / `~/.bash_login` / `~/.profile` の順に探し、**個人設定は最初に見つかった 1 つだけ**を読む
- 非ログインの対話的シェルは `~/.bashrc`(全体版は `/etc/bash.bashrc`)。`~/.bash_logout` **はログアウト時**
- 全体設定を足すのは `/etc/profile.d/` に `.sh` を 1 枚。`/etc/skel` **は新規ユーザー作成時の複写だけ**で既存ユーザーには効かない(`useradd -m` / `-k`)
- `env` は環境変数だけ、`set` は変数も関数も全部。`export` で環境変数に格上げ、`unset` で削除(関数は `unset -f`)。代入のイコール前後に空白を入れない
- 検索パスの追加は `export PATH=$PATH:/opt/bin` のように**既存の値を残す**。カレントディレクトリは既定で含まれないので `./script.sh` と書く
- エイリアスは引数を取れない → 引数が要るなら `function`。関数の定義は `function 名 { }` / `名() { }` / `function 名() { }` の 3 通りとも有効
- 変数を現在のシェルに残したいなら `source ファイル` または `. ファイル`(実行だと子シェルで消える)
- シバンは**必ず 1 行目・絶対パス**。`bash script.sh` で起動するとシバンは効かない
- スクリプトの定位置は `/usr/local/bin`。**シェルスクリプトの SUID はカーネルに無視される**ので、権限委譲は `sudo` で行う
- **終了ステータスは 0 が成功**。`echo $?` で確認、`exit 1` で異常終了、`set -e` で失敗時に即終了

- `[ ]` の内側には**空白必須**。数値比較は `-eq` `-lt` `-ge` など(不等号を使うとリダイレクトになる)、文字列は `=` と `!=`、空判定は `-z` / `-n`
- `;` は無条件、`&&` **は成功時、**`||` **は失敗時**。`&` 1 つはバックグラウンド実行で別物
- コマンド置換は `$(コマンド)` (古い形はバッククォート)。`$((式))` は算術演算で別物
- 終わりの記号: `if` は `fi`、`case` は `esac`、`for` / `while` / `until` は `done`。case の区切りは `;;`
- `while` は条件が真の間、`until` は真になるまで。`read` は変数名にドル記号を付けない。`seq` の 3 引数は **開始・増分・終了**
- `exec` **は子プロセスを作らず現在のシェルを置き換える**。コマンドを書かずにリダイレクトだけ指定すると以降の出力先が変わる
- 異常時だけ管理者へ通知するなら `コマンド || echo 本文 | mail -s 件名 root`
- `-x` **は展開後の実行内容、**`-v` **は読み込んだ行そのまま、**`-n` **は構文チェックのみ**

サンプルはここまでです

続き(第2章〜巻末)は、LPIC-1 102 問題集のご購入で、頻出度別ノート・暗記用ノートと合わせて3点すべてダウンロードできます。

LPIC-1 102試験 学習用テキスト(無料サンプル)

版
2026年7月版(LPI公式 LPIC-1 Exam 102 Objectives Version 5.0 準拠)

発行
IT資格道場(https://www.shikaku-doujou.com)

お問い合わせ
info@shikaku-doujou.com