

# シェルスクリプト

筑波大学医学医療系精神医学

根本 清貴

# 本日の内容

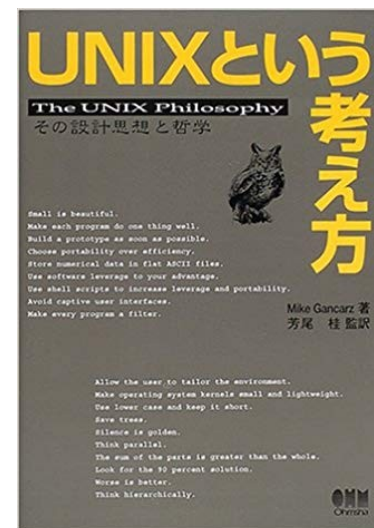
- 第1部:UNIX系OSのお作法を知る(兼予習)
- 第2部:画像解析ソフトのコマンドに慣れる
- 第3部:テキストを処理してみる
- 第4部:明日から使えるシェルスクリプトを知る
  - 1. 繰り返し
  - 2. 条件分岐

# 勉強会のルール

- ターミナルでタイプするものは、青色 (0000cc) で表示
  - 例: `$ fslhd V_ID001.nii.gz`
  - \$はタイプする必要はない
- スクリプトに記載する内容は緑色 (007e00) で表示
- コマンドやスクリプトではあるが、タイプしなくていいものは、紫色 (9933ff) で表示
- #以降は、解説でありタイプする必要はない
- 「フォルダ」=「ディレクトリ」
  - Linux/UNIXは、「ディレクトリ」を好む

# UNIXの思想

- 小さいものは美しい
- 各プログラムが一つのことをうまくやるようにせよ
- できる限り早く原型(プロトタイプ)を作れ
- 効率よりも移植しやすさを選べ
- 単純なテキストファイルにデータを格納せよ
- ソフトウェアを槌子(てこ)として利用せよ
- 効率と移植性を高めるためにシェルスクリプトを利用せよ
- 拘束的なユーザーインターフェースは作るな
- 全てのプログラムはフィルタとして振る舞うようにせよ



# シェルスクリプトのお約束

- 一番最初にシェバン shebang行を入れる
  - `#!/bin/bash` (Bash)
  - shebang: set of circumstances
- コメントをつける
  - shebang行以外の `#` から始まる行はコメント
  - コメントを入れておけば何をしたかったのか思い出せる
- 拡張子は `.sh` がおすすめ
  - シェルスクリプトとわかる
- 作成した後に `chmod 755` を忘れない
  - 実行権限をつける

# スクリプト作成の流れ

- 1例を処理する時の流れ(コマンドの羅列)を書き出し、原型とする
- 原型に対し、入力ファイルにより変化するところを同定する
- 変化するところを変数とし、これらを「処理内容」とする
- 入力画像を変数にセットするコマンドを記載する
- 繰り返し (`for`, `while`) や制御 (`if`) を組み入れる
- シェバン行を最初に書く

# スクリプトに役立つコマンドと変数

- コマンドなど

- `echo` 文字列を表示する
- `set -x` スクリプトの動作を表示する
- `$ ( )` コマンドの結果を変数に代入する
- ``` バッククォート; 上記とほぼ同じ

- 変数

- `"$@"` 指定した引数全部という意味
- `$#` 引数の数
- `$0` コマンド名
- `$1` コマンドの後に指定した1番目の引数
- `$?` コマンドの終了状態

# 画像解析スクリプトでよく使うコマンド

- 繰り返し
  - **for** 引数の数だけ繰り返し
  - **while** ある条件を満たす限り繰り返し
- 条件分岐
  - **if**
- 条件評価
  - **test**



# for

- 同じ「処理内容」を「引数」に対して繰り返し行う
- これは一番役立つ

for 変数 in 引数

do

処理内容

done

# FA画像作成スクリプト

- 拡散MRIデータから、FA画像を自動で作成したい
  - 必要なステップ
    - 渦電流補正 (`eddy_correct`)
    - マスク作成 (`bet`)
    - FA画像作成 (`dtifit`)

# 入力ファイルのファイル名確認

```
$ cd ~/shell_practice/scripts
```

```
$ ls D*
```

D\_subj1.bval

D\_subj2.bval

D\_subj1.bvec

D\_subj2.bvec

D\_subj1.nii.gz

D\_subj2.nii.gz

- D\_subj1 と D\_subj2 の2例に対して同じ処理を繰り返したい

# スクリプト作成の流れ

- 1例を処理する時の流れ(コマンドの羅列)を書き出し、原型とする
- 原型に対し、入力ファイルにより変化するところを同定する
- 変化するところを変数とし、これらを「処理内容」とする
- 入力画像を変数にセットするコマンドを記載する
- 繰り返し (`for`, `while`) や制御 (`if`) を組み入れる
- シェバン行を最初に書く

# gen\_fa.txt

- D\_subj1 を処理する時の一連のコマンドを、  
gen\_fa.txt というファイルに準備
  - "Generate FA" の気持ちでつけたファイル名
- まずは、このファイルの内容を確認

```
$ cat gen_fa.txt
```

# gen\_fa.txt

入力ファイル:D\_subj1.nii.gz

```
eddy_correct D_subj1 D_subj1_ecc 0
fslroi D_subj1_ecc D_subj1_b0 0 1
bet D_subj1_b0 D_subj1_brain -f 0.3 -m
dtifit --data=D_subj1_ecc \
      --out=D_subj1 \
      --mask=D_subj1_brain_mask \
      --bvecs=D_subj1.bvec \
      --bvals=D_subj1.bval
```

- これを一般化するためには、どこを変更すればよい？

# gen\_fa.txt

入力ファイル:D\_subj1.nii.gz

```
eddy_correct D_subj1 D_subj1_ecc 0
fslroi D_subj1_ecc D_subj1_b0 0 1
bet D_subj1_b0 D_subj1_brain -f 0.3 -m
dtifit --data=D_subj1_ecc \
      --out=D_subj1 \
      --mask=D_subj1_brain_mask \
      --bvecs=D_subj1.bvec \
      --bvals=D_subj1.bval
```

- 赤文字の部分が入力ファイルによって変わる場所

# スクリプト作成の流れ

- 1例を処理する時の流れ(コマンドの羅列)を書き出し、原型とする
- 原型に対し、入力ファイルにより変化するところを同定する
- 変化するところを変数とし、これらを「処理内容」とする
- 入力画像を変数にセットするコマンドを記載する
- 繰り返し (`for`, `while`) や制御 (`if`) を組み入れる
- シェバン行を最初を書く



# 変数の設定

- 共通するところを変数として設定する
- 今の場合、入力画像は拡散MRI画像なので、**dmri** とする
- 変数を使用する場合には、**\$dmri** となる
- 変数の後に文字列をつなげられるように、**\${dmri}** とする
- **D\_subj1** を **\${dmri}** として全部置換したい
- そのコマンドは→ **sed** !
- **\$ sed -e 's/D\_subj1/\${dmri}/g'** で全部置換できる

# sed で一括置換

```
$ sed -e 's/D_subj1/${dmri}/g' gen_fa.txt
```

```
eddy_correct ${dmri} ${dmri}_ecc 0
```

```
fslroi ${dmri}_ecc ${dmri}_b0 0 1
```

```
bet ${dmri}_b0 ${dmri}_brain -f 0.3 -m
```

```
dtifit --data=${dmri}_ecc \
      --out=${dmri} \
      --mask=${dmri}_brain_mask \
      --bvecs=${dmri}.bvec \
      --bvals=${dmri}.bval
```

- リダイレクションしないことで、これから変更する内容のプレビューができる
- 間違っていないことを確認したうえで、リダイレクションでファイルに書き出す

# リダイレクションで出力

```
$ sed -e 's/D_subj1/${dmri}/g' gen_fa.txt > gen_fa.sh
```

```
$ cat gen_fa.sh
```

```
eddy_correct ${dmri} ${dmri}_ecc 0
```

```
fslroi ${dmri}_ecc ${dmri}_b0 0 1
```

```
bet ${dmri}_b0 ${dmri}_brain -f 0.3 -m
```

```
dtifit --data=${dmri}_ecc      \  
      --out=${dmri}            \  
      --mask=${dmri}_brain_mask \  
      --bvecs=${dmri}.bvec      \  
      --bvals=${dmri}.bval
```

# スクリプト作成の流れ

- 1例を処理する時の流れ(コマンドの羅列)を書き出し、原型とする
- 原型に対し、入力ファイルにより変化するところを同定する
- 変化するところを変数とし、これらを「処理内容」とする
- 入力画像を変数にセットするコマンドを記載する
- 繰り返し (for, while) や制御 (if) を組み入れる
- シェバン行を最初を書く

# 入力画像を変数にセット

- 今、`D_subj1` を `${dmri}` に置換した
- 言い換えれば、`D_subj1` を 変数 `dmri` に代入することとなる

`dmri=D_subj1`

- このためには、画像から拡張子がとれていれば便利
- FSLには拡張子をとるコマンド `remove_ext` が準備されている

`$ remove_ext D_subj1.nii.gz`

`D_subj1`

# コマンドの結果を変数に代入

- **\$ ( )** により、コマンドの結果を変数に代入できる
  - バッククオート ` も使われるが、\$ ( ) が推奨されている
- 今、`remove_ext` により拡張子を取り除いたものを変数 `dmri` に代入したい
- その際のスクリプトへの記載は以下の通り  
`dmri=$(remove_ext D_subj1.nii.gz)`
- ここからさらに、`D_subj1.nii.gz` も変数で表示したい
  - このためには、`for` 文の理解が必要

# スクリプト作成の流れ

- 1例を処理する時の流れ(コマンドの羅列)を書き出し、原型とする
- 原型に対し、入力ファイルにより変化するところを同定する
- 変化するところを変数とし、これらを「処理内容」とする
- 入力画像を変数にセットするコマンドを記載する
- 繰り返し (**for**, **while**) や制御 (**if**) を組み入れる
- シェバン行を最初に書く

# for文

for 変数 in 引数

do

処理内容

done



# for文の第1行の理解

- **for 変数 in 引数** の意味は
  - 「引数」を次々に「変数」にいれるという意味
  - 変数はしばしば **f** が使われる (**file**)
  - 引数にはしばしば **"\$@"** が使われる
    - 「指定した引数すべて」という意味
- 今、変数 **f** に入るものは、画像ファイル
  - 今書こうとしているスクリプトの  
**dmri=\$(remove\_ext D\_subj1.nii.gz)**  
の **D\_subj1.nii.gz** を **\$f** に置き換えられる

# (参考) for文の第1行の例

- **for f in "\$@"**

- ユーザーが指定した「引数」を次々に変数 f にいれていく
- ユーザーはプログラムを実行する時に、処理したいものを明示する
- メリット: 処理したいファイルなどを選択できる
- デメリット: 完全自動化はできない

- **for f in \***

- プログラムを実行するワーキングディレクトリの中にあるすべてのファイル/ディレクトリを次々に変数 f にいれていく
- メリット: ディレクトリに処理したいファイルだけあるときには、完全自動化ができる(プログラムを実行するだけ)
- デメリット: 処理したくないファイルが入っているときには、エラーとなる

# (参考) for文の第1行の例

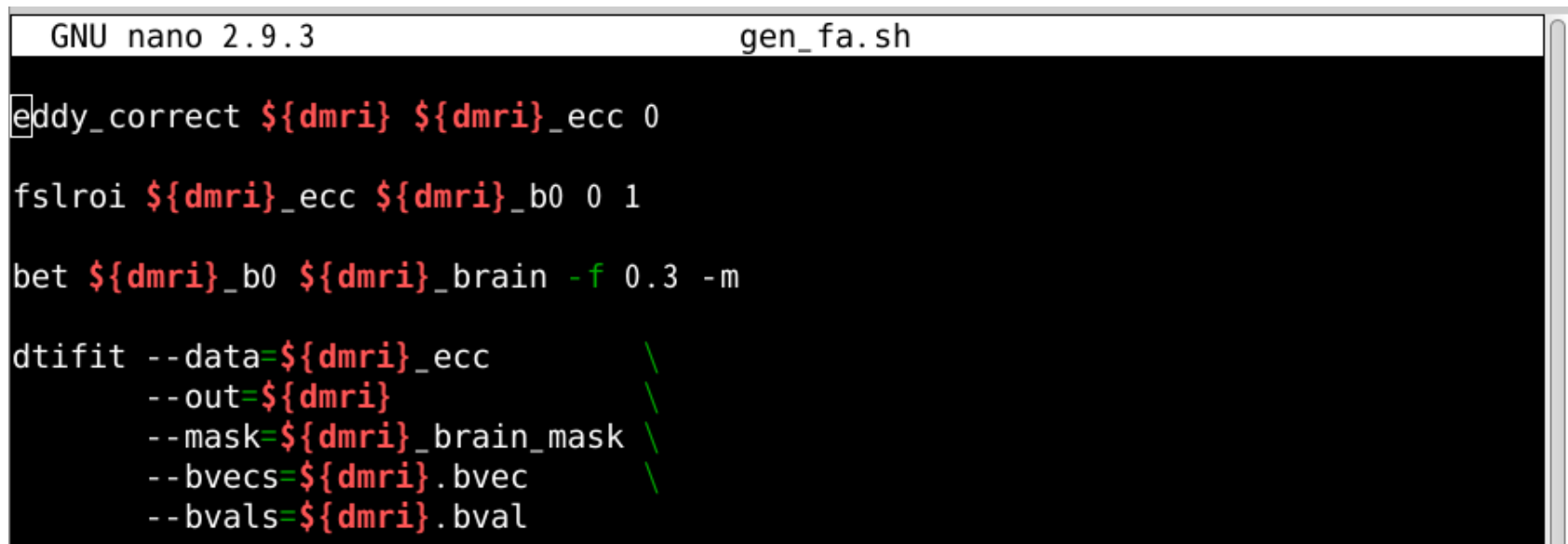
- **for f in D\*.nii.gz**
  - プログラムを実行するワーキングディレクトリの中にある「D ではなく .nii.gz で終わるファイル」を次々に変数 f にいれていく
  - メリット: ルールにマッチするファイルだけ処理できる
  - デメリット: ルールにマッチしていないファイルには応用できない
- **for dir in \$(ls -F | grep /)**
  - `ls -F` はディレクトリの最後に `/` をつけて表示するオプション
  - `ls -F | grep /` によりディレクトリだけが表示される
  - `$(ls -F | grep /)` により、ワーキングディレクトリの中にあるディレクトリを次々に変数 dir にいれていく

# nano でファイルを編集

- ターミナルから開けるエディタ、nano を用いてファイルを編集

**\$ nano gen\_fa.sh**

- マウスは一切使えない。↑ ↓ ← → で移動



```
GNU nano 2.9.3 gen_fa.sh

eddy_correct ${dmri} ${dmri}_ecc 0

fslroi ${dmri}_ecc ${dmri}_b0 0 1

bet ${dmri}_b0 ${dmri}_brain -f 0.3 -m

dtifit --data=${dmri}_ecc \
      --out=${dmri} \
      --mask=${dmri}_brain_mask \
      --bvecs=${dmri}.bvec \
      --bvals=${dmri}.bval
```

# for文の挿入

- 赤文字部分を追加

```
for f in "$@"
do
dmri=$(remove_ext $f)
eddy_correct ${dmri} ${dmri}_ecc 0
fslroi ${dmri}_ecc ${dmri}_b0 0 1
bet ${dmri}_b0 ${dmri}_brain -f 0.3 -m
dtifit --data=${dmri}_ecc \
      --out=${dmri} \
      --mask=${dmri}_brain_mask \
      --bvecs=${dmri}.bvec \
      --bvals=${dmri}.bval
done
```

# スクリプト作成の流れ

- 1例を処理する時の流れ(コマンドの羅列)を書き出し、原型とする
- 原型に対し、入力ファイルにより変化するところを同定する
- 変化するところを変数とし、これらを「処理内容」とする
- 入力画像を変数にセットするコマンドを記載する
- 繰り返し (`for`, `while`) や制御 (`if`) を組み入れる
- シェバン行を最初を書く

# Shebang とコメントの挿入、整形

- スクリプトにするためには、最初の行に `#!/bin/bash` を追加
- コメントを `#` で追加
- `for` の中であることをわかりやすくするために `for` の中身は半角スペース2文字～4文字でインデント
- 第2行に `set -x` を書いておくと、スクリプトの実行の際に何が起きているかを知ることができる(デバッグに有用)

# スクリプトの仕上げ

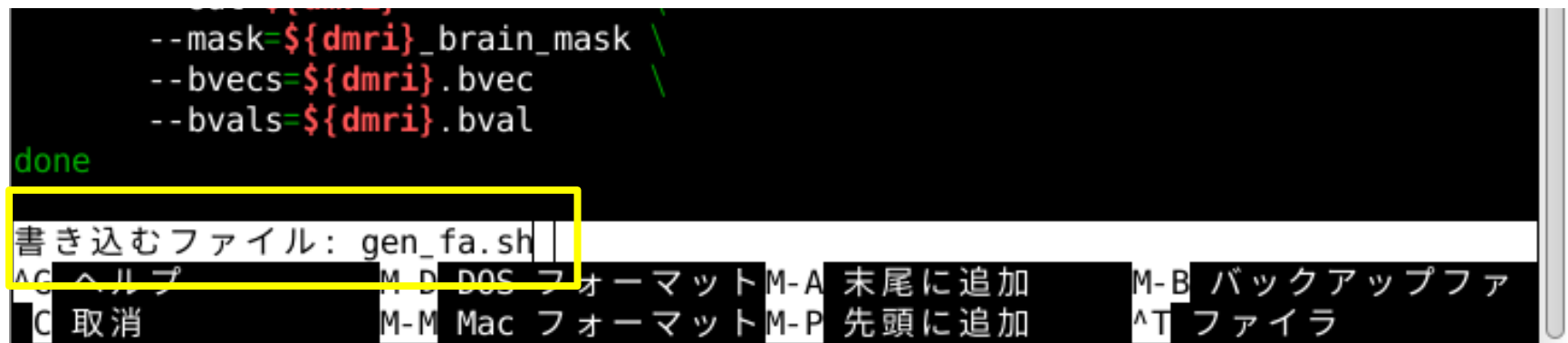
- 赤文字をさらに追加

```
#!/bin/bash
set -x
for f in "$@"
do
    dmri=$(remove_ext $f)
    # eddy current correction
    eddy_correct ${dmri} ${dmri}_ecc 0
    # extract b0 image
    fslroi ${dmri}_ecc ${dmri}_b0 0 1
    # generate mask
    bet ${dmri}_b0 ${dmri}_brain -f 0.72 -m
    # generate FA
    dtifit --data=${dmri}_ecc \
           --out=${dmri} \
           --mask=${dmri}_brain_mask \
           --bvecs=${dmri}.bvec \
           --bvals=${dmri}.bval
done
```



# スクリプトの保存

- nano の保存は、**^O** **Ctrl** + **O**
- nano で、**^** は **Ctrl** を意味する
- 「書き込むファイル: gen\_fa.sh」を確認し、Enter
- macOSの場合は**"File Name to Write: gen\_fa.sh"**



```
--mask=${dmri}_brain_mask \
--bvecs=${dmri}.bvec \
--bvals=${dmri}.bval
done
書き込むファイル: gen_fa.sh
```

^G ヘルプ    M-D DOS フォーマット    M-A 末尾に追加    M-B バックアップファ  
^C 取消    M-M Mac フォーマット    M-P 先頭に追加    ^T ファイラ

# 実行権限の付与

- nano を Ctrl + X で終了
- chmod で実行権限を付与

```
$ chmod 755 gen_fa.sh
```

# スクリプトの実行

- スクリプトは、パスが通っていないスクリプトは、パスを指定してプログラムを実行させる必要がある
- 今作成したプログラムがあるディレクトリはパスが通っていない
- パスの指定は簡単で、`./` をプログラムの前に書くだけ
- パスにおいて、`.` は「カレントディレクトリ」の意味
- `./gen_fa.sh` は「カレントディレクトリにある `gen_fa.sh` を実行しなさい」という意味になる
- `./` と `gen_fa.sh` の間には空白は入れてはいけない←パスだから

# スクリプトの実行

- 引数を指定しないでスクリプトを実行する

```
$ ./gen_fa.sh
```

- 何も起こらない
  - このスクリプトは、引数を指定するものだから
  - `for f in "$@"`

- 引数を指定してスクリプトを実行する

```
$ ./gen_fa.sh D_subj?.nii.gz
```

- `D_subj?.nii` の `?` はワイルドカード(任意の一文字)
- 今は、`subj1` か `subj2` なので、`?` とした
- 実行される #15分程度処理に時間がかかる

# 【演習】for文の読解(1)

- 次のスクリプトで行われる3つの処理を述べてください。なお、`unzip` は zipファイルを展開するコマンドです

```
#!/bin/bash
mkdir archive
for f in *.zip
do
    unzip $f
    mv $f archive
done
```

# 【演習の回答】for文の読解(1)

```
#!/bin/bash      #Shebang
mkdir archive    #archiveディレクトリを作成
for f in *.zip   #zip ファイルを変数 f に次々と代入
do
    unzip $f      #zipファイルの展開
    mv $f archive #展開が終わった zipファイルを archiveに移動
done
```

- このスクリプトは、あるディレクトリの中にあるzipファイルを自動で展開する
- 3つの処理は、
  - "archive"ディレクトリの作成
  - zipファイルの展開
  - 展開が終わったzipファイルの"archive"への移動
- `example` に `auto_unzip.sh` として保存してある

# 【演習】for文の読解(2)

- 次のスクリプトで行われる3つの処理を述べてください。なお、`zip -r` は ディレクトリをまるごと圧縮してzipファイルにするコマンドです

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

```
mkdir archive
```

```
for dir in $(ls -F | grep / | sed -e 's:/::')
```

```
do
```

```
    zip -r ${dir}.zip $dir
```

```
    mv ${dir}.zip archive
```

```
done
```

# 【演習の回答】for文の読解(2)

```
#!/bin/bash
mkdir archive
for dir in $(ls -F | grep / | sed -e 's:/::')
do
    zip -r ${dir}.zip $dir
    mv ${dir}.zip archive
done
```

- このスクリプトは、ディレクトリ毎にzipファイルとして圧縮する
- 3つの処理は
  - "archive"ディレクトリの作成
  - カレントディレクトリにあるディレクトリをひとつずつディレクトリ名がついた zipファイルとして圧縮
  - zipファイルを "archive"ディレクトリに移動
- example に auto\_zip.sh として保存してある



```
ls -F | grep / | sed -e 's:/::'
```

```
$ cd ~/shell_practice
```

```
$ ls -F
```

```
UnixIntro/  mri/  pair/  scripts/  textprocessing/  zip/
```

```
$ ls -F | grep /
```

```
UnixIntro/
```

```
mri/
```

```
pair/
```

```
scripts/
```

```
textprocessing/
```

```
zip/
```

```
$ ls -F | grep / | sed -e 's:/::'
```

```
UnixIntro
```

```
mri
```

```
pair
```

```
scripts
```

```
textprocessing
```

```
zip
```

# zip と unzip

- ディレクトリ `dirA` を `dirA.zip` という名前で圧縮したい時

```
$ zip -r dirA.zip dirA
```

– 「`dirA`を用いて `dirA.zip` を作りなさい」

- `fileA.zip` を展開したい時

```
$ unzip fileA.zip
```

# 【参考】for文のワンライナー

- コマンドラインにおいて、セミコロンは改行と同じ意味
- セミコロンを使うことで、コマンドを1行で記載できる

```
for f in *.zip
```

```
do
```

```
    unzip $f
```

```
done
```

は以下で書き表せる(ワンライナーという)

```
for f in *.zip; do unzip $f; done
```

- doの後はセミコロンが不要というのがミソ



# while

- ある「条件」が満たされている限り「処理内容」を繰り返す

while 条件

do

処理内容

done

# while の条件

- よく使われる条件は2種類
- **test** コマンドによる条件
  - **while** [ 条件 ]
  - 条件が成立する間、処理を繰り返す
    - 例: 画像一覧出力スクリプト
- **read** コマンドによる条件
  - **while read** 変数
  - テキストを1行ずつ変数に読み込み、テキストがある間、処理を繰り返す

# test

- **test** コマンドは、`[ ]` であらわし、中に評価式を書く。`[ ]` の前後は、必ず半角スペースが必要

`[ $num -lt 1 ]`      # \$num は1未満か？

**test** の結果は、0 か 1 か

- 0 → 条件を満たしている
- 1 → 条件を満たしていない
- この値は、**\$?** という特殊変数におさめられる

# test に使う評価式

|                            |        |
|----------------------------|--------|
| -lt: less than             | $<$    |
| -gt: greater than          | $>$    |
| -eq: equal                 | $=$    |
| -ne: not equal             | $\neq$ |
| -le: less than or equal    | $\leq$ |
| -ge: greater than or equal | $\geq$ |



# test の実際

```
$ num=3
```

#変数 num に3を代入

```
$ [ $num -lt 5 ]
```

# \$num は 5未満か？

```
$ echo $?
```

#評価結果が \$? に入る

0

```
$ [ $num -ge 5 ]
```

# \$num は 5以上か？

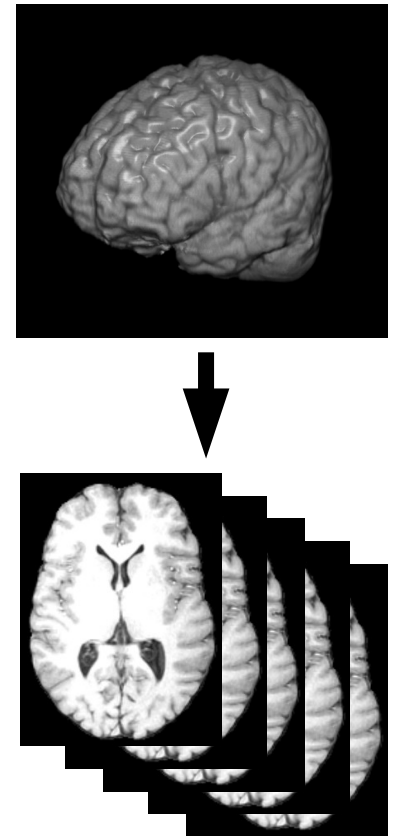
```
$ echo $?
```

#評価結果が \$? に入る

1

# nifti画像→png画像書き出しスクリプト

- 構造画像を axial 画像にして1スライスずつ png ファイルとして書き出す
- 戦略:
  1. nifti画像から拡張子を取り除き、変数 `id` に代入する
  2. nifti 画像から、z軸方向のスライス枚数(`dim3`)を読み取り、変数 `z` に代入する
  3. 変数 `num` を定義し、`$num=1` とする(初期化)
  4. `$num` が `$z` より小さいか評価する
  5. `$num` が `$z` より大きければ終了
  6. `$z` より小さいならば、`$num` のスライス番号の画像を png画像として出力する
  7. `$num` に 1 を足す
  8. 4.に戻る



# fslval

- FSLのツールで、nifti画像のヘッダー情報の数値を出力する

`fslval nifti画像 キーワード`

- 今、z軸の枚数は ヘッダーの `dim3` に入っているため、`im1.nii.gz` の `dim3` を知りたかったら次のようになる

`$ fslval im1.nii.gz dim3`

`($ fslinfo im1.nii.gz | awk '/^dim3/{print $2}'` と同義だがどちらが簡単か一目瞭然)

# 変数の増やし方

- 今、変数 num を設定し、それをひとつずつ増やしていく
- bash では、`$(( ))` の中に書くと四則演算は行ってくれる
- `num=$(( num + 1 ))`

で現行の `num` に 1 を足したものを新たな `num` として代入する

`$ num=4` #変数 num に4を代入

`$ echo $num` #4

`$ num=$(( num + 1 ))` #1を足したものを代入

`$ echo $num` #5

# slicer

- FSLのツールのひとつで、nifti画像からある水平断、冠状断、矢状断のスライスをpng画像として切り出せるコマンド
- 水平断は -z で出力できる
- スライス番号(n) は -n で指定する

```
$ slicer nifti画像 -z -n png画像
```

# これらをまとめる

- `brain_slice.sh` にスクリプトの基本が書かれている

```
$ pwd          #~/shell_practice
```

```
$ ls
```

```
$ cd scripts
```

```
$ less brain_slice.sh
```

# brain\_slice.sh

```
id=$(remove_ext $1)      #引数から拡張子を除き、変数 $id に代入
z=$(fslval $id dim3)      #fslvalで dim3を抜き出しzに代入

num=1                     #numを初期値1としてセット
while [ $num -le $z ]     # $num が $z 以下ならば以下を実行
do
    slicer $id -z -${num} ${id}_tmp_${num}.png
    # $num 番目のスライスを ${id}_tmp_${num}.png として保存
    num=$(( num+1 ))      # num に 1を足す
done
```

# brain\_slice.sh の実行

- `~/shell_practice/UnixIntro` でスクリプトを実行する

```
$ pwd                #~/shell_practice/scripts
```

```
$ cp brain_slice.sh ../UnixIntro  #スクリプトをコピー
```

```
$ cd ../UnixIntro
```

```
$ ls
```

```
$ chmod 755 brain_slice.sh  #スクリプトに実行権限を付加
```

```
$ ls
```

```
$ ./brain_slice.sh im1.nii.gz  #スクリプトを実行
```

```
$ ls
```

```
(Lin4Neuro) $ xdg-open .  #ファイルマネージャーを起動
```

```
(macOS) $ open .          #Finderを起動
```



# xdg-open / open

- Linuxでは **xdg-open**、MacOS では **open** はターミナルからGUIを操るのに有用
- コマンドの後にカレントディレクトリを意味する **.** をつけると、ファイルマネージャー / Finder を起動することができる

```
$ xdg-open .
```

```
$ open .
```

- その他、登録されている拡張子ならば、(ExcelファイルやWordファイルなど) "**xdg-open/open** ファイル名"でそのまま起動できる

# brainslicer.sh

- **brain\_slice.sh** に機能を追加したもの
  - 横8列に並べる
  - 引数を指定しないとエラーになる
  - スクリプトのひとつひとつにコメントをつけてある
- **scripts/example** ディレクトリに入っている

# while read

- テキストファイルを1行ずつ設定した変数に読み込んでいき、処理をする
- 空白行があるとエラーになるため、個人的には、パイプで sed と組み合わせて空行を削除することを習慣としている
- 「ファイルの各行に文字列があれば処理内容を処理する」

```
sed '/^$/d' ファイル | while read 変数
```

```
do
```

```
    処理内容
```

```
done
```

# sed ' /^\$/d '

- sed ' /パターン文字列/d ' は各行に「パターン文字列」があったらその行全体を削除する
- sed ' /^\$/ ' の ^\$ は、「『行頭』と『行末』が連続している行」という意味 → 「改行しかない空白行」を意味する
- while read で行を読み込む時に、空白行があるとエラーになるため、この手順を踏むことでエラーを回避する

# 画像振り分けスクリプト

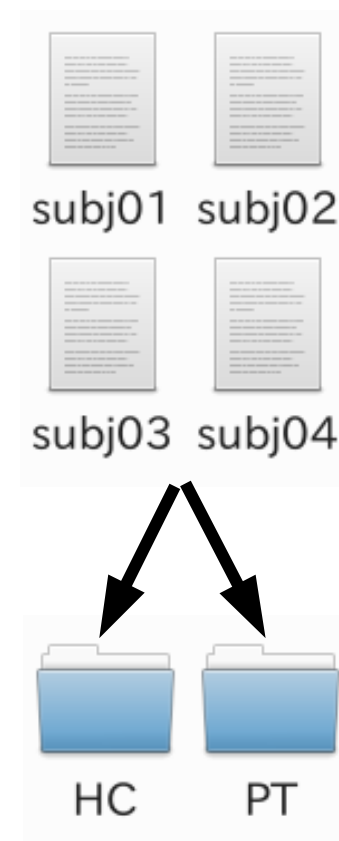
- テキストファイルにIDとグループを書いておくことで、自動でIDをグループに振り分けていく
- ヒューマンエラーを最小にできる
- テキストファイルの例

**subj01 HC**

**subj02 PT**

**subj03 PT**

**subj04 HC**



# 振り分けスクリプトの戦略

1. 引数として、既にIDとグループが書いてあるテキストを指定する
2. テキストを 変数 `$1` にセットし、`sed` を使って空白行を削除する
3. テキストを `read` で1行読み込み、第1フィールドを 変数 `id` に、第2フィールドを 変数 `category` にセットする
4. `$id` に合致するファイルを `$category` に合致するディレクトリにコピーする
5. 元のファイルは `original` ディレクトリに移動する

# 【演習】file\_sort.sh

```
$ pwd                                #~/shell_practice/UnixIntro  
$ cd ../scripts/                    #~/shell_practice/scripts  
$ ls  
$ less file_sort.sh
```

# file\_sort.sh

```
mkdir HC PT original    #ディレクトリを作成
sed '/^$/d' $1 | \       #テキストの空行を削除
while read id category   #テキストの第1フィールドをidに、
                        #第2フィールドをcategoryに代入
do
    cp *${id}* $category #条件にあうものをコピー
    mv *${id}* original  #originalに移動
done
```



# file\_sort.sh の実行

```
$ chmod 755 file_sort.sh    #実行権限を付加
$ cd sorting                #sortingに移動
$ ls                        #ファイル一覧を表示
$ less id_group.txt         #id_group.txtを確認
```

```
$ ../file_sort.sh id_group.txt
```

#スクリプトを実行。スクリプトは上のディレクトリにあるので、../ になる

```
$ ls                        #カレントディレクトリのファイル一覧を表示
$ ls HC                    #HCディレクトリのファイル一覧を表示
$ ls PT                    #PTディレクトリのファイル一覧を表示
$ ls original              #originalディレクトリのファイル一覧を表示
```

# 【演習】while文の読解

- param.txt に次のような記載があります

```
subj01 0.35
```

```
subj02 0.45
```

```
subj03 0.60
```

- 次のスクリプトは何をするスクリプトでしょうか？

```
sed -e '/^$/d' param.txt | while read id frac  
do  
    bet ${id} ${id}_brain -f ${frac}  
done
```

# 【演習の回答】while文の読解

- param.txt に次のような記載があります

```
subj01 0.35
```

- 次のスクリプトは何をするスクリプトでしょうか？

```
sed -e '/^$/d' param.txt | while read id frac
do
    bet ${id} ${id}_brain -f ${frac}
done
```

- bet で脳抽出する際のパラメータを個々に設定するスクリプト
- param.txt の第1フィールドが \$id, 第2フィールドが \$frac に代入され、bet の -f で設定される数値に示されている
- while を使うことで、個々で調整しなければならない作業も、事前にリストを作成しておくことで自動化することができる



# if文

- 条件によって、処理内容を変更する
- 「もし条件式1を満たすならば、処理内容1を実行、もし条件式2を満たすならば、処理内容2を実行、「それ以外は、処理内容3を実行」
- 条件2以降は省略できる

```
if 条件式1 ; then
```

```
    処理内容1
```

```
elif 条件式2 ; then
```

```
    処理内容2
```

```
else
```

```
    処理内容3
```

```
fi
```

# if を用いた引数のチェック

- 先程作成した `gen_fa.sh` スクリプトを走らせる際に、引数を指定していなかったら、エラーが出るようにしたい
- `if`文と`test ( [ ] )`を使ってこれを実現できる

# gen\_fa.sh に追加

```
$ cd ..    #~/shell_practice/scripts
```

```
$ nano gen_fa.sh
```

- 赤文字部分を追加 追加したら、`Ctrl` + `O` で保存, `Ctrl` + `X` で終了

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

```
set -x
```

```
if [ $# -lt 1 ] ; then
```

```
    echo "Error! No file is specified!"
```

```
    echo "Usage: $0 file(s)"
```

```
    exit 1
```

```
fi
```

```
for f in "$@"
```

```
do
```

# testの復習

- `test`は、`[ ]` であらわす
- `[ ]`の前後は、必ず半角スペースが必要

`[ $# -lt 1 ]`

`$#` 引数の数を表す特殊な変数

`-lt` less thanの意味

`[ $# -lt 1 ]`は、

「引数の数が1未満（つまり0）」という意味



# gen\_fa\_script.sh の実行

- 早速、あえてエラーを出してみる

**\$ ./gen\_fa.sh** (引数指定せず)

- どんなエラーが出るか？

# \$0 がスクリプトのパスに置き換わっている

```
$ ./gen_fa.sh
```

```
Error! No file is specified!
```

```
Usage: ./gen_fa.sh file(s)
```

スクリプトに記載した内容

```
echo "Error! No file is specified!"
```

```
echo "Usage: $0 file(s)"
```

```
exit 1 #exit 1は「エラーで終了になった」ことをシェルに伝える
```

# 【演習】if文の読解

- 自分の疲労度を 0-100 とし、`fatigue` という変数に代入するとします
- 現在のあなたの状況ではスクリプトの結果はどうなるでしょうか？

```
if [ $fatigue -gt 80 ]; then
```

```
    echo "おつかれさまでした。今日はゆっくりお休みください"
```

```
elif [ $fatigue -gt 50 ]; then
```

```
    echo "復習できるエネルギーは残っていそうですね"
```

```
else
```

```
    echo "まだまだ余裕ですね。課題を差し上げましょうか?"
```

```
fi
```

# 【演習の解説】if文の読解

```
if [ $fatigue -gt 80 ]; then
    echo "おつかれさまでした。今日はゆっくりお休みください"
elif [ $fatigue -gt 50 ]; then
    echo "復習できるエネルギーは残っていそうですね"
else
    echo "まだまだ余裕ですね。課題を差し上げましょうか?"
fi
```

- if は上から順に条件分岐をしていく。この場合は、まず、疲労度が80より高いかを判定する。高ければ「おつかれさまでした」を、そうでなければ次の条件式に進む
- 次は、50より高いかを判定する。80より上は既に条件分岐しているので、ここは51～80の人が該当することとなる
- 上から順に判断されることは頭に入れておくとい

# 質疑応答