Perl入学式 第3回

配列操作/ハッシュ/リファレンス編

諸注意

- 会場について
- 写真撮影について

紹介

- 講師・サポーター紹介
- 自己紹介

今日の流れ

- 前回の復習
- 配列の関数（2）
- ハッシュ
- リファレンス

用語説明

シジル

$, @, %, & といった、変数名の前に付く記号を指します。

リスト

```
('this', 'is', 'list', 1, 2, 3);
```

- このように、数値や文字列などをコンマ(,)で区切って並べて、括弧で囲ったものを「リスト」と呼びます

前回までに学んできたこと

- 変数
- 標準入力
- 演算子
- if, else 文
- for 文
- 配列
- qwショートカット
- 配列の関数（1）

復習問題

practice.md の `sum.pl` と `even_or_odd.pl` をやってみましょう。

配列の関数（2）

そもそも配列とは

前回、皆さんは配列の基本を勉強しました。

でも、そもそも配列ってどんなときに使うものでしょう？

配列の用途

- 要素間の順序関係を表現したい場合 (例: 待ち行列)
- 要素の単純な集まりとして表現したい場合 (例: 集合)

こうした操作をしたいとき、配列は役に立ちます

配列を扱う

配列を自在に操るためには、配列の要素を自由に

- 追加
- 取り出し

できる必要があります。

要素の追加

- push
- unshift

要素の取り出し

- pop
- shift

追加と取り出しの関係

- push / pop
- unshift / shift

push / pop

```
my @array = ('Alice', 'Bob');
push @array, 'Chris';      #=> ('Alice', 'Bob', 'Chris')
my $element = pop @array; #=> ('Alice', 'Bob')
print $element;            #=> "Chris"
```

push

[a][b][c] <=[d] 末尾に要素を追加する

pop

[a][b] =>[c] 末尾の要素を取り出す

unshift / shift

```
my @array = ('Alice', 'Bob');
unshift @array, 'Chris';    #=> ('Chris', 'Alice', 'Bob')
my $element = shift @array; #=> ('Alice', 'Bob')
print $element;             #=> "Chris"
```

unshift

[d]=> [a][b][c] 先頭に要素を追加する

shift

[a]<= [b][c] 先頭の要素を取り出す

練習問題 (1/2)

- 次の処理をする `array_pop_shift.pl` を作りましょう。
  1. ('Alice', 'Bob', 'Chris') という配列を作ってください。
  2. 1の配列から 'Chris' を取り出し、出力してください。
  3. 1の配列から 'Alice' を取り出し、出力してください。

練習問題 (2/2)

- 次の処理をする `array_push_unshift.pl` を作りましょう。
  1. ('Alice', 'Bob', 'Chris') という配列を作ってください。
  2. 1の配列の末尾に `Diana` を追加し、 ('Alice', 'Bob', 'Chris', 'Diana') という配列を作ってください。
  3. 2の処理に続けて、配列の最初に `Eve` を追加し、 ('Eve', 'Alice', 'Bob', 'Chris', 'Diana') という配列を作ってください。
  4. 3の配列をfor文で出力してみましょう。

reverse

reverse はリストを逆順に並べ替えて、そのリストを返します。

```
my @lang = qw(perl php ruby python java go);
my @reversed = reverse @lang;
print "@reversed"; #=> go java python ruby php perl
```

reverse

連番を逆順で配列に格納したいときにも便利です。

```
my @array = reverse ( 1 .. 5 ); #=> ( 5, 4, 3, 2, 1 )
```

sort

sort はリストをルール順に並べ替えて、そのリストを返します。
sortのみ, あるいは sort { $a cmp $b } @array と書くと, 「文字列」としてソートします

```
my @lang = qw(perl php ruby python java go);
my @sorted_lang = sort @lang; #=> go java perl php python ru

my @num = ( 5, 200, 40, 3, 1 );
my @sorted_num = sort @num; #=> (1, 200, 3, 40, 5)
```

sort

sort { $a <=> $b } @arrayと書くと,「数値」としてソートします

```
my @array = ( 5, 200, 40, 3, 1 );
my @sorted = sort {$a <=> $b} @array; #=> (1, 3, 5, 40, 200)
```

変数$aと$bはsortで使うために予約されているので, **sort以外で使ってはいけません**

ハッシュ

ハッシュとは

- ハッシュはPerlのデータ構造の1つで、「連想配列」とも呼ばれます。
- 配列と同じく、値の格納・取り出しができます。
- しかし配列とは異なり、「数値」ではなく「名前」を使って、格納する「値」を指定します。
  - 要素を指定するための「名前」を "key"、key によって指定された「値」を "value" と呼びます。

ハッシュの基本

以下の要素を持つ人物をハッシュで表現してみましょう。

- 名前: Alice
- 職業: Programmer
- 所属: PerlEntrance

ハッシュの基本

もしこれを配列で表現するなら、以下のようになるでしょう。

```
my @user = ('Alice', 'Programmer', 'PerlEntrance');
print $user[0]; # 最初の要素が表示される  => "Alice"
print $user[1]; # 2番目の要素が表示される => "Programmer"
print $user[2]; # 3番目の要素が表示される => "PerlEntrance"
```

- しかしこの場合は、添字 0 が名前、1 が職業、といった条件を覚えておく必要があります。
- これは非常に面倒ですね。

ハッシュの基本

同じ内容をハッシュで表現すると、このようになります。

```
my %user = (
    name        => 'Alice',
    job         => 'Programmer',
    affiliation => 'PerlEntrance'
);
print $user{name}; # nameが表示される => "Alice"
print $user{job};  # jobが表示される => "Programmer"
print $user{affiliation}; # affiliationが表示される => "PerlEn
```

- 添字（key）から値（value）を連想できるため、配列を使うよりわかりやすいですね。

ハッシュの基本

ハッシュには他にも利点があります。

以下のような配列があったとき、

```
my @user = ('Name', 'Job', 'Affiliation');
```

もし突然、要素の順番が入れ替わってしまったらどうなるでしょう？

```
my @user = ('Affiliation', 'Job', 'Name');
```

添字と値の関係が変わってしまうため、影響するコードを書き換えなくてはいけません。
しかしハッシュを使えば、値は添字の数値（順番）ではなく、名前（key）で対応づけられているので、そういった問題が起こりません。

ハッシュを使ってみよう

では、ハッシュを作ってみましょう。

```
my %hash = (
    name => 'Alice',
    age  => 16
);
```

- ハッシュは % を使って定義します。
- => はファットコンマ演算子と呼ばれ、コンマと同等の役割を果たします。
- => の左にkey（名前）、右にvalue（値）を書くことで、関係性が明らかになります。
- keyは文字列として解釈されるので、クォートの必要はありません。
- 1つのハッシュ内に文字列や数値が混在しても構いません。

ハッシュを使ってみよう

ハッシュの要素にアクセスし、取り出してみましょう。
ハッシュにアクセスするときは、添字として波括弧 {} を使います。
{}にkeyを入れることで、対応するvalue（値）を取り出せます。

```
my %hash = (
    name => 'Alice',
    age  => 16,
);
print $hash{name}; #=> 'Alice'
print $hash{age}; #=> 16
```

ハッシュを使ってみよう

ハッシュに新たな要素を代入してみましょう。
取り出すときと同様に、{key} を使います。

```
$hash{job} = 'Programmer'; # 新たな値'Programmer'を
                           # key'job'で代入
```

これで、ハッシュの中身は以下のようになりました。

```
my %hash = (
    name => 'Alice',
    age  => 16,
    job  => 'Programmer',
);
```

取り出してみましょう。

```
print $hash{job}; #=> 'Programmer'
```

ハッシュを使ってみよう

```
my %hash = (
    name => 'Alice',
    age  => 16,
    job  => 'Programmer',
);
```

- 質問: ハッシュの中身を全部一度に見たい場合はどうするの？
- 回答: Data::Dumperモジュールを使います。これについては後で解説します。

ハッシュを使ってみよう

じつはハッシュは配列の一種なので、このように書くこともできます。

```
my %hash = ('name', 'Alice', 'age', 16);
print $hash{name}; #=> "Alice"
print $hash{age};  #=> "16"
```

実際にこのように書くことは少ないですが、配列の一種であることは覚えておくとよいでしょう。

練習問題

- 次の処理をする `hash_profile.pl` を作りましょう。
  1. 自分の名前 (name)、年齢 (age)、好きな食べ物 (food) を key にしたハッシュ %my_profile を作ってください。
  2. key である `name`, `age`, `food` を使って、それぞれの value を出力してください。

ハッシュの便利な関数たち

ハッシュを便利に扱うための関数について説明します。

- keys
- values
- delete
- exists

keys

```
my %hash = (
    name        => 'Alice',
    job         => 'Programmer',
    affiliation => 'PerlEntrance'
);
my @keys = keys %hash;
print "@keys\n";    #=> "name job affiliation"
```

keys関数は、ハッシュの key を配列にして返します。

- ただし、この時 key は順不同です。（※とても重要）
- つまり、ハッシュに書かれた順番で返ってくるとは限りません。
- よって、同じ順番で受け取りたい場合は、sort関数を使って並び替えましょう。

values

```
my %hash = (
    name        => 'Alice',
    job         => 'Programmer',
    affiliation => 'PerlEntrance'
);
my @values = values %hash;
print "@values\n"; #=> "Alice Programmer PerlEntrance"
```

values関数は、ハッシュの value を配列にして返します。

- value も順不同で返ります。

delete

```
my %hash = (
    name        => 'Alice',
    job         => 'Programmer',
    affiliation => 'PerlEntrance'
);
delete $hash{affiliation};

# この時、%hash は以下のようになっています
# %hash = ( name => 'Alice', job => 'Programmer' );
```

delete関数は、指定したハッシュの key と、それに対応する value を削除します。

exists

```
my %hash = (
    name        => 'Alice',
    job         => 'Programmer',
    affiliation => 'PerlEntrance'
);
if (exists $hash{job})  { print "exists" } #=> 'exists'
if (exists $hash{team}) { print "exists" } # 何も出てこない
```

existsは、指定したハッシュの key が存在するか確認します。

練習問題

- 次の処理をする `hash_func.pl` を作りましょう。
  1. keys関数を使って、`hash_profile.pl` で作ったハッシュのkeyをすべて出力してください。
  2. delete関数を使って、1で使ったハッシュから年齢(age)の要素を削除してください。
  3. exists関数を使って、年齢の要素が存在するか確認してください。削除されている場合は "Age is not exist." と表示するようにしてみましょう。

リファレンス

リファレンスとは

リファレンスとは、配列やハッシュで扱うようなデータそのもの（実体）ではなく、それを指し示すもの（参照）を扱う方法です。
リファレンスを使うと、複雑なデータ構造を表現できます。

「複雑なデータ構造」とは？

ここでは、配列の中に配列が入っていたり、その中にさらにハッシュが入っているような「複数の階層を持つデータ」を指します。

リファレンスの利点

たとえば、このような構造のデータを扱いたいときに、

```
animal
 ├─ dog
 │    ├─ shiba
 │    └─ bulldog
 └─ cat
```

配列を使えば1階層分の処理は可能ですが、

```
my @animal = ('dog', 'cat');
print $animal[1]; #=> cat
```

複数の階層をまたぐ処理はできません。

```
my @animal = (('shiba', 'bulldog'), 'cat');
print $animal[0][1]; #=> 'bulldog' を表示したいがエラー
```

しかし、リファレンスを使えばこれが可能になります。

リファレンスの作り方

変数のシジル（$, @, %）の前にバックスラッシュ (\) を置くことで
各種のリファレンスを作れます。

```
my $scalar     = 'scalar'; # スカラーを定義
my $scalar_ref = \$scalar; # スカラーのリファレンスを作成

my @array      = ( 'foo', 'bar', 'baz' ); # 配列を定義
my $array_ref  = \@array;   # 配列のリファレンスを作成

my %hash       = ( foo => 'bar' ); # ハッシュを定義
my $hash_ref   = \%hash;    # ハッシュのリファレンスを作成
```

リファレンスの簡単な作り方（1）

先ほどは、配列のリファレンスを作るために元の配列を作りました。

```
my @array      = ( 'foo', 'bar', 'baz' ); # リファレンスの元にな
my $array_ref  = \@array;
```

しかし、配列の要素を [] でくくると、直接配列のリファレンスを作れます。

```
my $array_ref = [ 'foo', 'bar', 'baz' ]; # 直接リファレンスを作
```

スカラー変数に代入している点に注意してください。
このように、元の配列を持たない配列リファレンスを無名配列と呼びます。
初めからリファレンスを作ることが目的の場合はこの方法が便利です。

リファレンスの簡単な作り方（2）

配列と同様、ハッシュの要素を {} でくくると、直接ハッシュリファレンスを作れます。

```
my $hash_ref = {
    foo => 'bar'
};
```

このように、元のハッシュを持たないハッシュリファレンスを無名ハッシュと呼びます。
初めからリファレンスを作ることが目的の場合はこの方法が便利です。

配列リファレンスを使う

では、以下のデータを配列リファレンスで処理してみましょう。

```
animal
 ├─ dog
 │    ├─ shiba
 │    └─ bulldog
 └─ cat
```

先ほどはこのように書いて失敗しましたが、

```
my @animal = (('shiba', 'bulldog'), 'cat');
```

配列のリファレンスを使えば、意図した構造でデータを格納できます。

```
my @dog = ('shiba', 'bulldog'); # 配列@dogを作る
my $dog_ref = \@dog;            # @dogをもとに配列リファレンス$d
my @animal = ($dog_ref, 'cat'); # $dog_refを含む配列@animalを作
```

無名配列を使って、一度にまとめて書くこともできます。

```
my @animal = (['shiba', 'bulldog'], 'cat');
```

配列リファレンスを使う

では、@animalからbulldogを取り出すにはどうすればよいでしょう？

まずは配列の復習をかねて@dogから要素を取り出してみましょう。

```
print $dog[1]; #=> bulldog
```

同様に、配列リファレンス$dog_refから要素を取り出すにはこのようにします。

```
print ${$dog_ref}[1]; #=> bulldog
```

なぜこのような書き方になるのでしょうか？

- じつは、配列のリファレンスを{}で囲んで先頭に@を付けると、元の配列として扱うことができます。
- つまり、@{$dog_ref}と@dogは等価です。
- よって、配列から要素を取り出すときと同様に、シジル@を$に変えて、添字[1]を付けることで、要素を取り出すことができるのです。

配列リファレンスを使う

```
1. 配列@dogを作る。
@dog = ('shiba', 'bulldog');

2. 複雑なデータ構造を扱えるようにするため、
配列@dogから配列リファレンス$dog_refを作る。
my $dog_ref = \@dog;

3. 配列リファレンス$dog_refを@{$dog_ref}とすることで、
@dogと同等に扱えるようにする。

4. @{$dog_ref}から要素`bulldog`を取り出す。
print ${$dog_ref}[1]; #=> bulldog
```

4のように、リファレンスから元のデータにアクセスすることを「デリファレンス」と言います。

配列リファレンスを使う

配列のデリファレンスは以下のように簡単に書くこともできます。

```
print $dog_ref->[1]; #=> bulldog
```

通常の配列から要素を取り出す形にアロー（->）を加えるだけですから、実際にはこの書き方を使う方が便利です。

```
print $dog[1];         # 通常の配列から要素を取り出す場合
print $dog_ref->[1]; # 配列リファレンスから要素を取り出す場合
```

配列リファレンスを使う

では、ここまでの知識を組み合わせて、@animalから要素を取り出してみましょう。

```
print ${$animal[0]}[1]; #=> bulldog
```

アロー（->）を使えばこのように書けます。

```
print $animal[0]->[1];  #=> bulldog
```

アローが添字に挟まれる場合は、省略してこのようにも書けます。

```
print $animal[0][1];    #=> bulldog
```

練習問題

```
animal
 ├─ dog
 │    ├─ shiba
 │    └─ bulldog
 ├─ cat
 │    ├─ mike
 │    └─ abyssinian
 └─ bird
      ├─ eagle
      └─ crow
```

- 次の処理をする array_ref.pl を作りましょう。
  1. 上記のデータ構造を表す配列リファレンス$animalを作ってください。
  2. その中からeagleを取り出してください。
  3. 余裕があれば、他の要素を取り出したり、要素を増やしたりしてみましょう。

ハッシュリファレンスを使う

以下のようなデータ構造があるときに、

```
animal
 └─ dog
        ├─ name: 'Taro'
        └─ color: 'brown'
```

ハッシュを使えば1階層分の処理は可能ですが、

```
my %dog = (
    name => 'Taro',
    color => 'brown',
);
print $dog{name}; #=> Taro
```

ハッシュリファレンスを使う

複数の階層をまたぐ処理はできません。

```
my %animal = (
    dog => (
        name => 'Taro',
        color => 'brown',
    ),
);
print $animal{dog}{name}; #=> 'Taro'を表示したいがエラー
```

しかし、リファレンスを使えばこれが可能になります。

ハッシュリファレンスを使う

ハッシュリファレンスを使えば意図した構造でデータを格納できます。

```
my %dog = (
    name => 'Taro',
    color => 'brown',
);
my $dog_ref = \%dog; # %dogをもとにハッシュリファレンス$dog_refを

my %animal = (
    dog_key => $dog_ref,
); # $dog_refをvalueとする%animalを作る
```

無名ハッシュを使って、一度にまとめて書くこともできます。

```
my %animal = (
    dog_key => {
        name => 'Taro',
        color => 'brown',
    },
);
```

ハッシュリファレンスを使う

では、%animalからTaroを取り出すにはどうすればいいでしょうか？

まずハッシュの復習をかねて%dogから要素を取り出してみましょう。

```
print $dog{name}; #=> Taro
```

同様に、ハッシュリファレンス$dog_refから要素を取り出すにはこのようにします。

```
print ${$dog_ref}{name}; #=> Taro
```

なぜこのような書き方になるのでしょうか？

- じつは、ハッシュリファレンスを{}で囲んで先頭に%を付けると、元のハッシュとして扱うことができます。
- つまり、%{$dog_ref}と%dogは等価です。
- よって、ハッシュから要素を取り出すときと同様に、シジル%を$に変えて、{key}を付けることで要素を取り出すことができるのです。

ハッシュリファレンスを使う

```
1. ハッシュ%dogを作る。
my %dog = ( name => 'Taro', color => 'brown',);

2. 複雑なデータ構造を扱えるようにするため、
ハッシュ%dogから配列リファレンス$dog_refを作る。
my $dog_ref = \%dog;

3. ハッシュリファレンス$dog_refを%{$dog_ref}とすることで、
%dogと同等に扱えるようにする。

4. %{$dog_ref}から要素`Taro`を取り出す。
print ${$dog_ref}{name}; #=> Taro
```

ハッシュリファレンスを使う

ハッシュのデリファレンスは以下のように簡単に書くこともできます。

```
print $dog_ref->{name}; #=> Taro
```

通常のハッシュから要素を取り出す形にアロー（->）を加えるだけですから、実際にはこの書き方を使う方が便利です。

```
print $dog{name};         # 通常のハッシュから要素を取り出す場合
print $dog_ref->{name}; # ハッシュリファレンスから要素を取り出す場
```

ハッシュリファレンスを使う

では、ここまでの知識を組み合わせて、%animalからTaroを取り出してみましょう。

```
print ${$animal{dog_key}}{name}; #=> Taro
```

アロー（->）を使えばこのように書けます。

```
print $animal{dog_key}->{name}; #=> Taro
```

アローが{key}に挟まれる場合は、省略してこのようにも書けます。

```
print $animal{dog_key}{name}; #=> Taro
```

練習問題

```
animal
 ├─ dog
 │    ├─ name: 'Taro'
 │    └─ color: 'brown'
 └─ cat
      ├─ name: 'Tama'
      └─ color: 'white'
```

- 次の処理をする hash_ref.pl を作りましょう。
  1. 上記のデータ構造を表すハッシュリファレンス$animalを作ってください。
  2. その中からwhiteを取り出してください。
  3. 余裕があれば、他の要素を取り出したり、要素を増やしたりしてみましょう。

リファレンスに代入する

配列やハッシュと同様、リファレンスにも新たな要素を代入できます。

```
animal
 ├─ dog
 │    ├─ shiba
 │    └─ bulldog
 └─ cat
```

上記のデータ構造におけるdogの3番目の要素にpoodleを加えたい場合は、このようにします。

```
my @animal = (['shiba', 'bulldog'], 'cat');
${$animal[0]}[2] = 'poodle'; # 新たな要素を代入
```

無名配列を使って、このように書くこともできます。

```
$animal[0]->[2] = 'poodle';
$animal[0][2] = 'poodle';
```

リファレンスに代入する

```
animal
 └─ dog
        ├─ name: 'Taro'
        └─ color: 'brown'
```

上記の構造におけるdogに、key: valueの関係がage: 3になる新たな要素を代入する場合はこのようにします。

```
my %animal = (
    dog => {
        name => 'Taro',
        color => 'brown',
    },
);
${$animal{dog}}{age} = 3; # 新たな要素を代入
```

無名ハッシュを使って、このように書くこともできます。

```
$animal{dog}->{age} = 3;
$animal{dog}{age} = 3;
```

リファレンスの中身を全部見る

データ構造の中身を出力したいとき、ここまではその要素の一部だけを取り出してきました。

しかし、中身を一度にすべて見たい場合はどうすればいいでしょうか？

ためしに、以下のハッシュリファレンス（無名ハッシュ）を普通にprintしてみましょう。

```
my $dog = {
    name => 'Taro',
    color => 'brown',
};
$dog->{age} = 3;

print $dog; #=> HASH(0x7f8aa2029380)
```

リファレンスの中身を全部見る

- `HASH(0x7f8aa2029380)` のような出力が得られたと思います。(括弧内の英数字は実行環境によって異なります)
- 今回の例はハッシュリファレンスなのでHASH()となっていますが、スカラーリファレンスならSCALAR()、配列リファレンスならARRAY()と表示されます。
- 初めに説明したように、リファレンスとはデータそのもの（実体）ではなく、それを指し示すもの（参照）を扱うものでした。
- じつはHASH() の括弧内の英数字は、実体が格納されているアドレス（場所の情報）です。
- print関数は実体を出力する働きをもつため、アドレスが表示されてしまうのです。

Data::Dumperを使う

このようなときは、Data::Dumper モジュールを使えばリファレンスの中身を一括表示（ダンプ）できます。

```
use Data::Dumper; # 最初にモジュールの使用を宣言
my $dog = {
    name => 'Taro',
    color => 'brown',
};
$dog->{age} = 3;
print Dumper($dog); # Data::Dumperを使ってprint
# 出力結果↓
# $VAR1 = {
#          'name' => 'Taro',
#          'color' => 'brown',
#          'age' => 3
#          };
```

この例のように、新たな要素を代入した後に中身を確かめたいときや、データの中身を把握していないときなどに役立つので覚えておきましょう。

最終問題

- 次の処理をする array_hash_ref.pl を作りましょう。
  1. ここまでに学んだ内容をもとに、配列リファレンスとハッシュリファレンスが混在するリファレンス$array_hash_refを作ってください。
     - 中身の要素は自由に設定してください。
     - 題材が思い浮かばない場合は、自分の家族や会社、あるいは好きなスポーツチームや、ここにいる講師やサポーターを設定してみましょう。
  2. その中から、配列リファレンスの要素と、ハッシュリファレンスの要素をどれでもよいので一つずつ出力してください。
  3. その後、Data::Dumper を使ってリファレンス全体の中身を出力してください。

復習問題

- https://github.com/perl-entrance-org/workshop-2016/blob/master/3rd/practice.md
  - 今回の内容を復習できる問題集です。
  - 不明点があれば、気軽にサポーターに質問してください。