情報メディア基盤ユニット用資料（2013 年 5 月 28 日分）

Processing 言語による情報メディア入門

座標変換 (続き) と関数 (その 1)

神奈川工科大学情報メディア学科　佐藤尚

プログラムが動かない - Σヽ(` д ´ ;)ノうぉぉぉ！となる前に

サンプルのプログラムを入力すると、上手く実行出来ないことがあります。その時に、チェックした方がよい点を挙げておきます。これ以外にも、原因があると思いますが、とりあえず気がついた点です。基本的には､「ちゃんとプログラムは入力されていますか？」､「ちゃんと正しい場所にデータはコピーされていますか？」です。

表 7-1 プログラムが動作しないときのチェックリスト

挙動	チェックポイントなど
なんだか動かない “The function 関数名 does not exist.” と表示される。	プログラムは正しく入力されていますか？特に、関数名は正しく入力されていますか？大文字と小文字は正しく区別されていますか？ Processing では大文字と小文字を区別します。
なんだか動かない “The field Component. 変数 is not visible.” “Cannot find anything named 変数名 ” ”Cannot find class or type named 名前 ” などと表示される。	プログラムは正しく入力されていますか？特に、変数名やデータ型の名称はは正しく入力されていますか？大文字と小文字は正しく区別されていますか？ Processing では大文字と小文字を区別します。
なんだか動かない “Syntax error, maybe a missing right parenthesis?” と表示さる。	どこかに ”)” を忘れていませんか？特に、黄色くなっている行の辺りです。
なんだか動かない “Unexpected token: 文字列 ” と表示される。	どこかに ”(”、"{" や "}" を忘れていませんか？特に、黄色くなっている行やその少し上の行の辺りです。
なんだか動かない “The method 関数名 (……” などと表示がされる。	引数同士の区切りが "," ではなく、"." になっていませんか？小数点(.)が "," になっていませんか？引数はあっていますか？特に、黄色くなっている行の辺りです。
なんだか動かない unexpected char: '\'” と表示される。	変数名や関数名などに全角文字を使っていませんか？プログラムの空白に全角スペースを利用していませんか？

人間は間違える生き物です。間違えなく入力したと思っても、普通は間違いがあります。本人は間違いなく入力したと思い込んでいるので、間違えを見つけることが出来ません。情報システムのデザインをするときには、人間は間違えるかもという視点を持ってデザインすることは重要です。
ところで、ファミコンとスーパーファミコンのカセットの違いを知っていますか？ 3.5 インチのフロッピーディスクの形を知っていますか？どちらも、古すぎる話でしょうか？

エディタ上で CTRL-t を（control キーと t キーを同時に）押すと、プログラムのインデントなどを自動でつけてくれます。

(と) の対応や { と } の対応はエディタ上で簡単にチェックが出来ます。この機能を上手く使って下さい。
例えば、(の辺りにカーソルを持って行くと、対応する) が、四角形で囲まれます。

```
void setup()
img = loadImage("test.jpg");
size(400,400);
}
```

ここです。

挙動	チェックポイントなど
なんだか動かない “Syntax error, maybe a missing semicolon?” などと表示されます。	セミコロン ; を忘れていませんか？黄色で表示されている部分またはその少し前にセミコロンを忘れている場所はありませんか？
なんだか動作がおかしい 指定した大きさのウインドウが開かない。	「void setup(){」の部分で、setup の綴りを間違えていませんか？
なんだか動作がおかしい ウインドウが灰色のままで、画像が表示されない。	「void draw(){」の部分で、draw の綴りを間違えていませんか？
途中でプログラムが止まる ”NullPointerExceptoin” と表示される。	loadImage 関数で指定している画像ファイル名は正しいですか？指定の場所に画像ファイルがありますか？ 通常、表示したい画像ファイルは Sketch > Show Sketch Folder で表示されるフォルダ内の data フォルダ内に保存します。
途中でプログラムが止まる ”A null PFont was passed …” などと表示される。	きちんとフォントデータが読み込まれていますか？
なんだか動作がおかしい **or** **途中でプログラムが止まる** ”Could not load font vlw ファイル名 .” と表示される。	loadFont で指定している vlw ファイルのファイル名は正しいですか？指定の場所に vlw ファイルが保存されていますか？ Tools > Create Font で vlw ファイルを作成していますか？ 通常、ｖｌｗファイルは Sketch > Show Sketch Folder で表示されるフォルダ内の data フォルダ内に置いておきます。

座標変換の続き

座標軸の移動の続きです。translate 関数や rotate 関数単体での動きはわかりやすいと思います。

translate 関数は、translate 関数の引数で指定された値分だけ、「現在の原点」を移動させます。この時に、移動方向は、「現在の座標軸」が基準となります。図 7-1 のように、黒い座標軸が「現在の座標軸」とすると、これを基準に移動を行います。「現在の座標軸」が傾いていれば、傾いた方向に移動することなります。

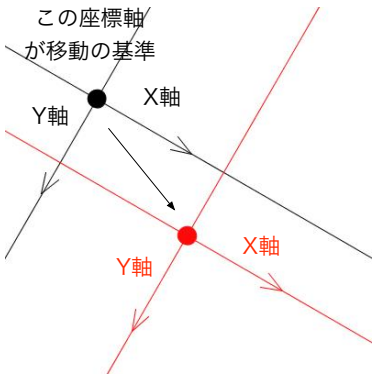


図 7-1 translate での座標軸の移動

rotate 関数も translate 関数と同じように、「現在の座標軸」を基準に回転を行います。回転の中心は、「現

在の原点」です。図 7-2 のように、黒い座標軸が「現在の座標軸」とすると、これを基準に回転を行います。

setup 関数と draw 関数を使ってプログラムを作成する際には、draw 関数の先頭では、「現在の座標軸」は初期状態（原点はウインドウの左上、水平に X 軸、垂直に Y 軸）となります。

translate 関数や rotate 関数は単体で用いるより、組み合わせて使用することが普通です。ある場所で、物体を回転させたい場合には、回転の中心としたい場所に、translate 関数を用いて「現在の原点」に移動させ、その後に、rotate 関数を使えば、好きな場所で物体を回転させることができます。

translate 関数や rotate 関数を使って「現在の座標軸」を動かしていくと、「現在の座標軸」を初期状態に移動させたいことがあります。これを行うのが、resetMatrix 関数です。この関数を実行すると、「現在の座標軸」が初期状態に戻ります。

また、「途中の座標軸」の状態を保存をしておきたいこともあります。Processsing では、どこかの変数に状態を保存するのでなく、行列スタックと呼ばれる場所に保存します。「現在の座標軸」を保存するのに使用するのが pushMatrix 関数で、「現在の座標軸」を保存されている座標軸の状態に戻す際に使用するのが popMatrix 関数です。pushMatrix 関数と popMatrix 関数はペアになって使用します。もし、ペアになって使用されていないと、直ぐにエラーが発生します。

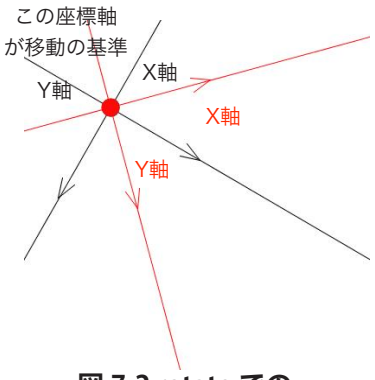


図 7-2 rotate での座標軸の移動

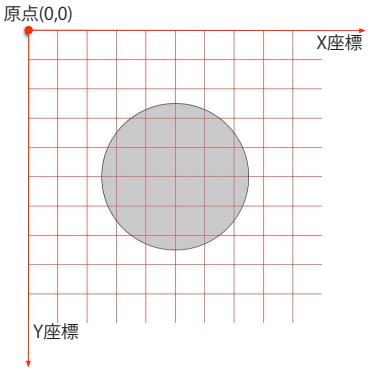


図 7-3 初期状態の座標軸

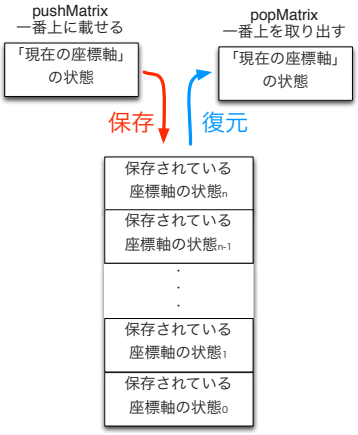


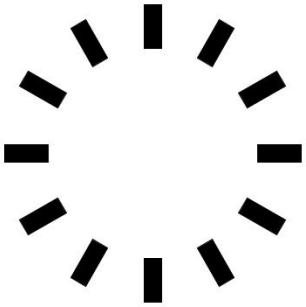
図 7-4 行列スタック

前回のサンプル 6-18 が、このことを利用して、マウスカーソルの周りで長方形を回転させています。

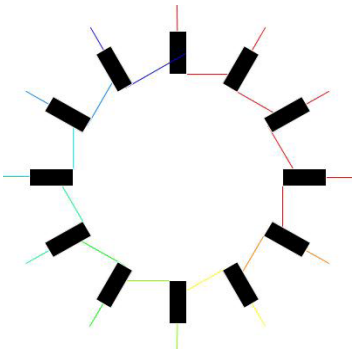
スタック (stack) は、コンピュータのプログラムでは良く出てくるデータを蓄えるための考え方です。最後に入れたデータ (push) が、最初に出てくる (pop) という動作をするようなものです。
学食などにある、トレーを沢山つんである山を想像すると良いかもしれません。push はデータを書いた紙をトレーの山の一番上に載せることに相当します。pop はトレーの山の一番上にあるトレーを取り除き、そこに書かれているデータを読み出すことに相当します。

translate と rotate を利用した例その 1 サンプル 7-1

```
size(400,400);
smooth();
stroke(0);
fill(0);
background(255);
// ウインドウの中心に「現在の座標軸」を移動
translate(width/2,height/2);
```



```
for(int i=0;i < 12;i++){
  pushMatrix(); //「現在の座標軸」をスタックの一番上に載せる
  rotate(radians(-90+30*i)); //「現在の座標軸」を回転させる
  translate(120,0); //「現在の原点」を移動させる
  rect(0,-10,50,20); // 長方形を描画する
  // 現在の座標軸」をスタックの一番上にある座標軸の状態に変更する
  popMatrix();
}
```



サンプル 7-1 は pushMatrix 関数や popMatrix 関数を使わなくても書くことが出来ます。

translate と rotate を利用した例その 1' サンプル 7-2

```
size(400,400);
smooth();
stroke(0);
fill(0);
background(255);
for(int i=0;i < 12;i++){
// ウインドウの中心に「現在の座標軸」を移動
  resetMatrix();
  translate(width/2,height/2);
  rotate(radians(-90+30*i)); //「現在の座標軸」を回転させる
  translate(120,0); //「現在の原点」を移動させる
  rect(0,-10,50,20); // 長方形を描画する
}
```

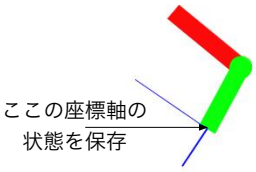
サンプル 7-1 は pushMatrix 関数と popMatrix 関数を使用しなくても、簡単に同じ動作をするプログラムを書くことができます。しかし、次の様なロボットアームのような動作をするプログラムでは、pushMatrix 関数と popMatrix 関数を使用することなくプログラムを作成することは困難です。

translate と rotate を利用した例その 2 サンプル 7-3

```
float angle;
color arm1,arm2,arm3;

void setup(){
  size(400,400);
  smooth();
  angle = 0;
  arm1 = color(255,10,10);
  arm2 = color(10,255,10);
  arm3 = color(10,10,255);
}
```

```
void draw(){
  background(255);
  translate(width/2,height/2);
  rotate(radians(angle));
  stroke(arm1);
  fill(arm1);
  rect(0,-10,100,20);
  translate(100,0);
  stroke(arm2);
  fill(arm2);
  ellipse(0,0,25,25);
  rotate(radians(2*angle));
  rect(0,-10,80,20);
  translate(80,0);
  stroke(arm3);
  pushMatrix();
  rotate(radians(6*minute()));
  strokeWeight(2);
  line(0,0,50,0);
  popMatrix();
  rotate(radians(6*second()));
  strokeWeight(1);
  line(0,0,100,0);
  angle += 0.1;
}
```



今週の演習問題で出題されているアナログ時計を作る際には、draw 関数の内部で

1. pushMatrix 関数を実行
2. 時間を表す針を描画
3. popMatrix 関数を実行
4. pushMatrix 関数を実行
5. 分を表す針を描画
6. popMatrix 関数を実行
7. pushMatrix 関数を実行
8. 秒を表す針を描画
9. popMatrix 関数を実行

のような順番で処理をおこなって行きます。

7 番の pushMatrix 関数と 9 番の popMatrix 関数は実行しなくても大丈夫です。

変数の有効範囲

情報メディア学科で、鈴木先生と言うと、鈴木浩先生のことを指します。情報工学科で、鈴木先生と言うと、鈴木孝幸先生のことを指します。この二人が同時にいる場所で、鈴木先生と言うと、どちらの鈴木先生を指しているのかわからなくなります。お父さんというと、家によって誰を指すのかが変わります。このように、ある名前がどこまで有効かを決めないと、混乱してしまいます。そこで、Processing などのプログラミング言語でも、変数の有効範囲の規則


どちらも鈴木先生

が決まっています。

変数の有効範囲により、次の 2 つの区別があります。

1. 大域変数（グローバル変数）
2. 局所変数（ローカル変数）

大域変数は基本的にプログラム中のどこでも利用することができます。一方、局所変数は、その変数を使える場所が限られている変数です。大域変数と局所変数の宣言の仕方に違いはなく、どの場所でその変数を宣言したかで決まります。

今までのサンプルでは、基本的にプログラムの先頭で変数の宣言を行ってきました。このようにプログラムの先頭で変数を宣言すると大域変数として扱われます。一方、関数内部の変数を使い始めたい場所で、変数宣言を行うと、局所変数として扱われます。局所変数は使える場所は、基本的に局所変数を宣言したブロック内部（{〜}）となります。

例えば、サンプル 7-4 では、プログラムの先頭で変数宣言を行っている int 型の変数 xPos は大域変数となります。また、「int x=xPos;」となっている int 型の変数 x は局所変数となります。そして、変数 x の有効範囲は、変数宣言を行ったブロックの内部の、変数宣言を行った以降の部分（赤色の文字の部分）となります。変数 xPos は大域変数なので、setup 関数の内部や draw 関数の内部の両方で利用することが出来ます。

この規則のことを、スコープ規則やスコープルールと呼ぶことがあります。

有効範囲とは、その変数が使える場所という意味です。

後で、メンバ変数というものが出てきます。

対応する { と } の間がブロックです。

Processing の変数の有効範囲に関する知識は、そのまま C++ 言語や Java 言語でも使えます。しかし、C++ 言語では少し異なる部分があります。C 言語では、もっと異なる部分が増えます。

大域変数と局所変数の例 1 サンプル 7-4

```
int xPos; // この変数は大域変数です。プログラム中のどこでも使えます。

void setup(){
  size(400,100);
  smooth();
  xPos = width;
}

void draw(){
  background(255);
  fill(128);
  stroke(0);
  int x = xPos; // この変数は局所変数です。有効範囲は赤色の部分のみ。
  while(x < width){
    ellipse(x,height/2,20,20);
    x += 10;
  }
  xPos--;
}
```

このサンプルは、for 命令を使っても書くことが出来ます。これを行ったのものがサンプル 7-5 です。このサンプルでは、変数 xPos は大域変数です。「for(int x=xPos;…」の部分で宣言している変数 x は局

所変数となります。この変数 x の有効範囲は「for(int x=xPos;…){ 〜 }」の部分（赤字の部分）になります。

大域変数と局所変数の例 2 サンプル 7-5

```
int xPos; // この変数は大域変数です。プログラム中のどこでも使えます。

void setup(){
  size(400,100);
  smooth();
  xPos = width;
}

void draw(){
  background(255);
  fill(128);
  stroke(0);
  for(int x=xPos;x < width;x += 20){ // 変数 x は局所変数です。
    ellipse(x,height/2,20,20);
  }
  xPos--;
}
```

for 命令の場合には、for 命令全体でブロックを作っているように動作となっています。ちょっと注意が必要かも知れません。

この 2 つのサンプルは同じ動作をするものですが、局所変数 x を宣言する場所が少し異なっているので、サンプル 7-4 は次の様に while 命令終了後に変数 x の値を表示させることが出来ますが、サンプル 7-5 では for 命令終了後に変数 x の値を表示させる命令を追加するとエラーとなります。

大域変数と局所変数の例 3 サンプル 7-4'

```
int xPos; // この変数は大域変数です。プログラム中のどこでも使えます。

void setup(){
  size(400,100);
  smooth();
  xPos = width;
}

void draw(){
  background(255);
  fill(128);
  stroke(0);
  int x = xPos; // この変数は局所変数です。有効範囲は赤色の部分のみ。
  while(x < width){
    ellipse(x,height/2,20,20);
    x += 10;
  }
  println(x);
  xPos--;
}
```

大域変数と局所変数の例 4 サンプル 7-5'

```
int xPos; // この変数は大域変数です。プログラム中のどこでも使えます。

void setup(){
  size(400,100);
  smooth();
  xPos = width;
}

void draw(){
  background(255);
  fill(128);
  stroke(0);
  for(int x=xPos;x < width;x += 20){ // 変数 x は局所変数です。
    ellipse(x,height/2,20,20);
  }
  println(x);
  xPos--;
}
```

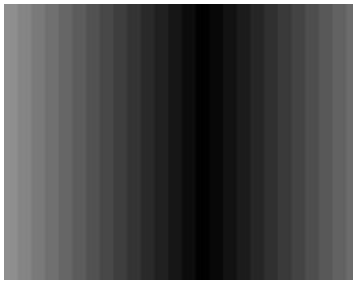
赤色の文字の部分が局所変数 x の有効範囲です。この場所は変数 x の有効範囲を出ているので、エラーメッセージ" The filed Componet.x is not visible." 表示されます。

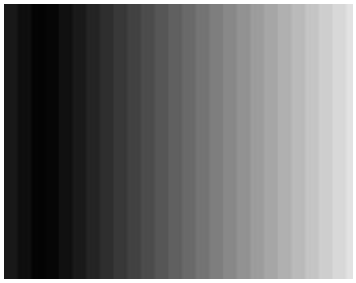
サンプル 7-6 では、大域変数は使用していませんが、局所変数 x と gray を使用しています。局所変数 gray は 2 箇所で宣言していますが、異なるブロックで宣言しています。従って、名前の混乱を引き起こすことがないので、このような使い方が可能です。

大域変数と局所変数の例 5 サンプル 7-6

```
void setup(){
  size(255,200);
  smooth();
}

void draw(){
  background(0);
  noStroke();
  int x= 0; // 局所変数、赤字の部分で有効
  while(x <= mouseX){// (1)
    int gray = mouseX-x; // この gray は、(1) の while 命令内で有効
    fill(gray);
    rect(x,0,10,height);
    x += 10;
  }
  while(x < width){  // (2)
    int gray = x-mouseX; // この gray は、(2) の while 命令内で有効
    fill(gray);
    rect(x,0,10,height);
    x += 10;
  }
}
```





局所変数 x と同じように、サンプル 7-6 の局所変数 gray をサンプル 7-7 のように使用するとエラーとなります。これは、局所変数 gray を宣言した場所が、while 命令のブロックの中なので、局所変数 gray の有効範囲は、このブロック（赤字の部分）に限られるためです。

大域変数と局所変数の例 5' サンプル 7-7

```
void setup(){
  size(255,200);
  smooth();
}
void draw(){
  background(0);
  noStroke();
  int x= 0;
  while(x <= mouseX){// (1)
    int gray = mouseX-x; // この gray は、(1) の while 命令内で有効
    fill(gray);
    rect(x,0,10,height);
    x += 10;
  }
  while(x < width){ // (2)
    fill(gray);   // 異なるブロックで宣言された変数なので使えない
    rect(x,0,10,height);
    x += 10;
  }
}
```

変数の有効範囲外になると、変数に記憶されていた情報は消えてしまいます。

この場所は変数 gray の有効範囲を出ているので、"Cannot find anything named "gray"" というエラーメッセージが表示されます。

サンプル 7-7 の while 命令をサンプル 7-8 のように for 命令に書きかえるとエラーとなります。

大域変数と局所変数の例 5''' サンプル 7-8

```
void setup(){
  size(255,200);
  smooth();
}
void draw(){
  background(0);
  noStroke();
  for(int x=0;x <= mouseX;x += 10){ // 変数 x は赤色の部分で有効
    int gray = mouseX-x;
    fill(gray);
    rect(x,0,10,height);
  }
// 変数 x は異なるブロックで宣言されているの無効
  for(;x < width;x+=10){
    int gray = x - mouseX;
    fill(gray);
    rect(x,0,10,height);
  }
}
```

この場所は変数 x の有効範囲を出ているので、エラーメッセージ "The filed Componet.x is not visible." 表示されます。

この場合には、サンプル 7-9 のように局所変数 x を宣言すると、エラーとならずサンプル 7-6 と同じ動作を行います。

大域変数と局所変数の例 5'''' サンプル 7-9

```
void setup(){
  size(255,200);
  smooth();
}

void draw(){
  background(0);
  noStroke();
  int x; // 変数 x は赤字の部分で有効
  for(x=0;x <= mouseX;x += 10){
    int gray = mouseX-x;
    fill(gray);
    rect(x,0,10,height);
  }
  for(;x < width;x+=10){
    int gray = x - mouseX;
    fill(gray);
    rect(x,0,10,height);
  }
}
```

{ ～ } で作られるブロックの中に新たなブロックを作ることが出来ます。サンプル 7-5 の for 命令を使用したサンプルやサンプル 7-6 の while 命令などが、その例になっています。入れ子になっているブロックで、外側のブロックで宣言した局所変数と同じ名前の局所変数を宣言することは出来ません。ですから、サンプル 7-10 はエラーとなります。

大域変数と局所変数の例 6 サンプル 7-10

```
void setup(){
size(360,360);
colorMode(HSB,359,99,99);
for(int x=0;x<width;x += 10){
  color c = color(x,99,99);
  for(int y=0;y < height;y += 10){
// 外側のブロックの局所変数と同じ名前の局所変数は宣言出来ない。
    color c = color(y,99,99);
    fill(c);
    rect(x,y,10,10);
  }
}
```

大域変数と同じ名前の局所変数を定義することは出来ます。ただし、同じ名前の局所変数が定義されているブロックでは、同じ名前の大域変数にアクセスするには、ちょっと工夫が必要です。
「this. 大域変数名」でアクセスすることが出来ます。詳しくは、説明しません。

"Duplicate local variable c" というエラーメッセージが表示されます。

関数の宣言（その 1）

コンピュータのプログラム作成では、同じような処理を行っている場合は、なるべくまとめて書くということが基本的な指針となっています。このような考え方から繰り返し処理の紹介を行ってきました。今度は、別の角度から同じような処理をまとめて書くということを行って行きます。

サンプル 7-11 は、ボディを表す長方形と 4 つのタイヤを表す長方形を描画することで、1 台の車のような形を表示するものです。

同じような処理をまとめるということの発想の裏には、モジュール化という発想があります。コンピュータの世界では、モジュール化という発想は非常に重要です。第二次世界大戦開始時には、戦車開発で遅れを取っていた米国は、このモジュール化という発想で、戦車を製作し、大量生産が可能になりました。だから勝てた？

1 台の車状の絵を表示その 1 サンプル 7-11

```
void setup(){
  size(400,400);
  smooth();
}

void draw(){
  background(255);
  rectMode(CENTER);
  float carX = width/2; // 車の中心の X 座標
  float carY = height/2;// 車の中心の Y 座標
  float carW = 120;     // 車の横幅
  float carH = carW/2.0; // 車の縦幅
  stroke(0);
  fill(150);
  rect(carX,carY,carW,carH); // ボディの描画
  fill(0);
  float tireW = carW/4.0;  // タイヤの横幅
  float tireH = carH/6.0;  // タイヤの縦幅
  // 4 つのタイヤの描画
  rect(carX-carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX+carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX-carW/4,carY+carH/2+tireH/2,tireW,tireH);
  rect(carX+carW/4,carY+carH/2+tireH/2,tireW,tireH);
}
```

たくさんの局所変数を宣言していますが、このほうがやっていることの意味がハッキリすると思います。

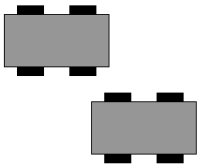
サンプル 7-11 では、車の中心座標を変数で与えているので、ちょっとした変更で、複数の車を表示するサンプルに書きかえることが出来ます。サンプル 7-12 は、2 台の車を表示するものです。

2 台の車状の絵を表示その 1 サンプル 7-12

```
void setup(){
  size(400,400);
  smooth();
}
```

```
void draw(){
  background(255);
  float carX = width/2; // 車の中心の X 座標
  float carY = height/2;// 車の中心の Y 座標
  float carW = 120;     // 車の横幅
  float carH = carW/2.0; // 車の縦幅
  rectMode(CENTER);
  stroke(0);
  fill(150);
  rect(carX,carY,carW,carH); // ボディの描画
  fill(0);
  float tireW = carW/4.0;  // タイヤの横幅
  float tireH = carH/6.0;  // タイヤの縦幅
  // 4 つのタイヤの描画
  rect(carX-carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX+carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX-carW/4,carY+carH/2+tireH/2,tireW,tireH);
  rect(carX+carW/4,carY+carH/2+tireH/2,tireW,tireH);

  carX = 100;// 車の中心の X 座標
  carY = 100;// 車の中心の Y 座標
  carW = 120;// 車の横幅
  carH = carW/2.0; // 車の縦幅
  stroke(0);
  fill(150);
  rect(carX,carY,carW,carH);// ボディの描画
  fill(0);
  tireW = carW/4.0;// タイヤの横幅
  tireH = carH/6.0;// タイヤの縦幅
// 4 つのタイヤの描画
  rect(carX-carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX+carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX-carW/4,carY+carH/2+tireH/2,tireW,tireH);
  rect(carX+carW/4,carY+carH/2+tireH/2,tireW,tireH);
}
```

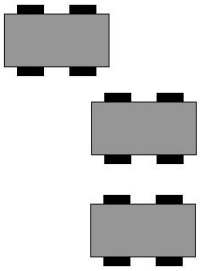


このサンプルは調子に乗ると、もっとたくさんの車を表示させることが出来ます。サンプル 7-13 では 3 台の車を表示させています。

3 台の車状の絵を表示その 1 サンプル 7-13

```
void setup(){
  size(400,400);
  smooth();
}
void draw(){
  background(255);
  float carX = width/2; // 車の中心の X 座標
  float carY = height/2;// 車の中心の Y 座標
  float carW = 120;     // 車の横幅
  float carH = carW/2.0; // 車の縦幅
```

```
  rectMode(CENTER);
  stroke(0);
  fill(150);
  rect(carX,carY,carW,carH); // ボディの描画
  fill(0);
  float tireW = carW/4.0;  // タイヤの横幅
  float tireH = carH/6.0;  // タイヤの縦幅
  rect(carX-carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX+carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX-carW/4,carY+carH/2+tireH/2,tireW,tireH);
  rect(carX+carW/4,carY+carH/2+tireH/2,tireW,tireH);
  carX = 100;// 車の中心の X 座標
  carY = 100;// 車の中心の Y 座標
  carW = 120;// 車の横幅
  carH = carW/2.0; // 車の縦幅
  stroke(0);
  fill(150);
  rect(carX,carY,carW,carH);// ボディの描画
  fill(0);
  tireW = carW/4.0;// タイヤの横幅
  tireH = carH/6.0;// タイヤの縦幅
  rect(carX-carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX+carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX-carW/4,carY+carH/2+tireH/2,tireW,tireH);
  rect(carX+carW/4,carY+carH/2+tireH/2,tireW,tireH);
  carX = mouseX;// 車の中心の X 座標
  carY = mouseY;// 車の中心の Y 座標
  carW = 120;// 車の横幅
  carH = carW/2.0; // 車の縦幅
  stroke(0);
  fill(150);
  rect(carX,carY,carW,carH);// ボディの描画
  fill(0);
  tireW = carW/4.0;// タイヤの横幅
  tireH = carH/6.0;// タイヤの縦幅
  rect(carX-carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX+carW/4,carY-carH/2-tireH/2,tireW,tireH);
  rect(carX-carW/4,carY+carH/2+tireH/2,tireW,tireH);
  rect(carX+carW/4,carY+carH/2+tireH/2,tireW,tireH);
}
```



サンプル 7-11 は、translate 関数を利用すると、サンプル 7-14 のように書き換えることが出来ます。車が固定したままだと面白くないので、マウスで移動できるようにもしてみました。

1 台の車状の絵を表示その 2 サンプル 7-14

```
void setup(){
  size(400,400);
  smooth();
}
```

```
void draw(){
  background(255);
  rectMode(CENTER);
  float carW = 120;     // 車の横幅
  float carH = carW/2.0; // 車の縦幅
  translate(mouseX,mouseY);  // 車の中心を「現在の原点」にする
  stroke(0);
  fill(150);
  rect(0,0,carW,carH); // ボディの描画
  fill(0);
  float tireW = carW/4.0;// タイヤの横幅
  float tireH = carH/6.0;// タイヤの縦幅
  // 4つのタイヤの描画
  rect(-carW/4,-carH/2-tireH/2,tireW,tireH);
  rect( carW/4,-carH/2-tireH/2,tireW,tireH);
  rect(-carW/4, carH/2+tireH/2,tireW,tireH);
  rect( carW/4, carH/2+tireH/2,tireW,tireH);
}
```

translate(carX,carY) で 点(carX,carY) に「現在の原点」を移動させているので、原点が車の中心と考えることができるので、タイヤの描画位置の計算が簡単になっています。

translate 関数を使って、2 台の車を表示するサンプルを作ってみます、この場合には、「現在の座標軸」の状態を記録するために、pushMatrix 関数と popMatrix 関数を使っています。

2 台の車状の絵を表示その 2 サンプル 7-15

```
void setup(){
  size(400,400);
  smooth();
}

void draw(){
  background(255);
  rectMode(CENTER);
  float carW = 120;     // 車の横幅
  float carH = carW/2.0; // 車の縦幅

  pushMatrix(); //「現在の座標軸」の状態を保存
  translate(mouseX,mouseY);  // 車の中心を「現在の原点」にする
  stroke(0);
  fill(150);
  rect(0,0,carW,carH); // ボディの描画
  fill(0);
  float tireW = carW/4.0;// タイヤの横幅
  float tireH = carH/6.0;// タイヤの縦幅
  // 4つのタイヤの描画
  rect(-carW/4,-carH/2-tireH/2,tireW,tireH);
  rect( carW/4,-carH/2-tireH/2,tireW,tireH);
  rect(-carW/4, carH/2+tireH/2,tireW,tireH);
  rect( carW/4, carH/2+tireH/2,tireW,tireH);
  popMatrix();
```

```
  pushMatrix(); //「現在の座標軸」の状態を保存
  translate(width/2,height/2);  // 車の中心を「現在の原点」にする
  stroke(0);
  fill(150);
  rect(0,0,carW,carH); // ボディの描画
  fill(0);
  tireW = carW/4.0;// タイヤの横幅
  tireH = carH/6.0;// タイヤの縦幅
  // 4 つのタイヤの描画
  rect(-carW/4,-carH/2-tireH/2,tireW,tireH);
  rect( carW/4,-carH/2-tireH/2,tireW,tireH);
  rect(-carW/4, carH/2+tireH/2,tireW,tireH);
  rect( carW/4, carH/2+tireH/2,tireW,tireH);
  popMatrix();
}
```

このようにサンプルを作ると、車の描画する部分の共通部分がハッキリしてきます。そこで、共通部分をまとめるのが関数と呼ばれる仕組みです。実は、今までも関数を使ってきました。つまり、setup や draw です。この場合には、setup 関数や draw 関数は、事前に Processing の側で使われることを知っている関数です。このようなもの以外に、プログラムを作る人が自由に関数を作ることが出来ます。自分なりの関数の作り方は、いくつかのパターンがあります。まずは、一番単純な関数の定義の仕方を紹介します。まず、関数を定義するためには、その関数の名前を決める必要があります。関数の名前のことを、関数名と呼びます。

英語では、関数のことを function と呼びます。

自分なりの関数を作ることを、関数を定義すると呼ぶことがあります。

簡単にいうと、setup や draw と同じです。

表 7-2 関数定義の仕方（その 1）

関数定義のパターン

void 関数名 (){
関数処理の内容を書きます。
変数なども使うことができます。
}

サンプル 7-15 を関数を使って書きかえてみます。車を描く部分を関数としてまとめるので、関数名は drawCar とします。定義した drawCar 関数を使いたいときには、使いたい部分で、「drawCar();」とするだけです。

2 台の車状の絵を表示その 2 サンプル 7-16

```
void setup(){
  size(400,400);
  smooth();
}
```

```
// drawCar 関数の定義
void drawCar(){
  float carW = 120;     // 車の横幅
  float carH = carW/2.0; // 車の縦幅
  rectMode(CENTER);
  stroke(0);
  fill(150);
  rect(0,0,carW,carH); // ボディの描画
  fill(0);
  float tireW = carW/4.0;// タイヤの横幅
  float tireH = carH/6.0;// タイヤの縦幅
  // 4 つのタイヤの描画
  rect(-carW/4,-carH/2-tireH/2,tireW,tireH);
  rect( carW/4,-carH/2-tireH/2,tireW,tireH);
  rect(-carW/4, carH/2+tireH/2,tireW,tireH);
  rect( carW/4, carH/2+tireH/2,tireW,tireH);
}

void draw(){
  background(255);
  pushMatrix();//「現在の座標軸」の状態を保存
  translate(mouseX,mouseY);// 車の中心を「現在の原点」にする
  drawCar(); // 定義した関数を呼び出す
  popMatrix(); //「現在の座標軸」を保存されている状態に戻す
  pushMatrix(); //「現在の座標軸」の状態を保存
  translate(width/2,height/2);// 車の中心を「現在の原点」にする
  drawCar(); // 定義した関数を呼び出す
  popMatrix();//「現在の座標軸」を保存されている状態に戻す
}
```

ここで定義した変数 carW、carH は、drawCar 関数内部でのみ使用できます。局所変数 carW と carH が定義されているブロックはどこでしょうか？

このサンプルをよく考えると、drawCar 関数を呼び出す際に、車を描く位置も指定できると、もっと簡潔にプログラムが書けるように考えられます。rect 関数や ellipse 関数では、図形を描く場所や大きさを引数として指定することが出来ます。これと同じことが自分で定義した関数でも出来れば、良いはずです。引数を使った関数定義の仕方は、次の様になります。

表 7-3 関数定義の仕方（その 2）

関数定義のパターン
void 関数名 (データ型名　引数名){
関数処理の内容を書きます。
変数なども使うことができます。
}
void 関数名 (データ型名 1　引数名 1,
データ型名 2　引数名 2…){
関数処理の内容を書きます。
変数なども使うことができます。
}

ここで出てくる引数名、引数名 1、引数名 2 などは、この関数の中だけで、有効な変数となります。
また、引数として宣言された変数は、関数内で局所変数として利用することが出来ます。

この引数付きの関数定義を利用してサンプル 7-16 を書きかえて見ます。非常にシンプルになったことがわかると思います。

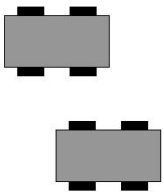
2 台の車状の絵を表示その 3 サンプル 7-17

```
void setup(){
  size(400,400);
  smooth();
}

// drawCar 関数の定義
void drawCar(float x,float y){
  float carW = 120;     // 車の横幅
  float carH = carW/2.0; // 車の縦幅
  rectMode(CENTER);
  pushMatrix();//「現在の座標軸」の状態を保存
  translate(x,y);// 車の中心を「現在の原点」にする
  stroke(0);
  fill(150);
  rect(0,0,carW,carH); // ボディの描画
  fill(0);
  float tireW = carW/4.0;// タイヤの横幅
  float tireH = carH/6.0;// タイヤの縦幅
  // 4 つのタイヤの描画
  rect(-carW/4,-carH/2-tireH/2,tireW,tireH);
  rect( carW/4,-carH/2-tireH/2,tireW,tireH);
  rect(-carW/4, carH/2+tireH/2,tireW,tireH);
  rect( carW/4, carH/2+tireH/2,tireW,tireH);
  popMatrix();//「現在の座標軸」を保存されている状態に戻す
}

void draw(){
  background(255);
  drawCar(mouseX,mouseY); // 定義した関数を呼び出す
  drawCar(width/2,height/2); // 定義した関数を呼び出す
}
```

float 型の変数 x と y は、drawCar 関数の内部だけで有効な変数となります。



引数付きの関数を呼び出すときには、少し動作が複雑になります。サンプル 7-17 で、「drawCar(mouseX,mouseY)」が実行されると、mouseX の値が drawCar 関数の引数 x に、mouseY の値が drawCar 関数の引数 y に、それぞれコピーされます。このコピーが終わった後に、drawCar 関数で指定されている処理の実行が始まります。

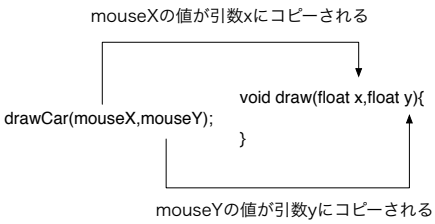


図 7-5 関数呼び出し時の引数のコピー

この drawCar 関数を使った、別のサンプルを載せておきます。このサンプル 7-18 では、自動車が移動していきます。また、サンプル 7-18 では、大域変数と同じ局所変数（引数）を使っています。あま

り良い習慣ではないと思いますが、大域変数名と同じ名前の局所変数を定義することが出来ます。その局所変数が定義されているブロックの中では、その局所変数が優先されますので、大域変数の値をアクセスすることは出来ません。

裏技 (this. 大域変数名) を使うとアクセスすることが出来ます。

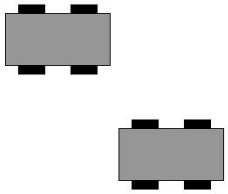
移動する車 サンプル 7-18

```
int x;

void setup(){
  size(400,400);
  smooth();
  x = 0;
}

// drawCar 関数の定義
void drawCar(float x,float y){
  float carW = 120;     // 車の横幅
  float carH = carW/2.0; // 車の縦幅
  rectMode(CENTER);
  pushMatrix();//「現在の座標軸」の状態を保存
  // この変数 x は drawCar 関数の引数 x を指します。
  translate(x,y);// 車の中心を「現在の原点」にする
  stroke(0);
  fill(150);
  rect(0,0,carW,carH); // ボディの描画
  fill(0);
  float tireW = carW/4.0;// タイヤの横幅
  float tireH = carH/6.0;// タイヤの縦幅
  // 4 つのタイヤの描画
  rect(-carW/4,-carH/2-tireH/2,tireW,tireH);
  rect( carW/4,-carH/2-tireH/2,tireW,tireH);
  rect(-carW/4, carH/2+tireH/2,tireW,tireH);
  rect( carW/4, carH/2+tireH/2,tireW,tireH);
  popMatrix();//「現在の座標軸」を保存されている状態に戻す
}

void draw(){
  background(255);
  drawCar(x,height/3); // 定義した関数を呼び出す
  drawCar(2*x,2*height/3);// 定義した関数を呼び出す
  x = (x+1) % width;
}
```



剰余演算 %（余りを求める）を使って、車の繰り返し移動を実現しています。
あることを行うプログラムには、色々なやり方があります。コンピュータに指示するやり方のことをアルゴリズム (algorithm) と呼んでいます。

もう一つの関数の使い方のサンプルを示します。サンプル 7-19 はウインドウの真ん中を左右にボールが移動し、壁にぶつかると反射するというものです。

移動するボール サンプルその 1 7-19

```
int xPos;
int speed;
int radius;

void setup(){
  size(400,200);
  smooth();
  xPos = width/2;
  speed = -1;
  radius = 20;
}

void draw(){
  background(255);
  // ボールを移動させる
  xPos = xPos+speed;
  // ボールの壁での反射処理を行う
  if((xPos+radius) > width){
    speed = -1;
    xPos = width-radius;
  }else if((xPos-radius) < 0){
    speed = 1;
    xPos = radius;
  }
  // ボールを描く
  stroke(0);
  fill(127);
  ellipse(xPos,height/2,2*radius,2*radius);
}
```

サンプル 7-19 は、draw 関数の中にすべての処理を書いています。このように、この程度の小さなプログラムでは、1 つの関数の中にすべての処理を書いてしまっても、大きな問題は発生しません。人間はあまり記憶力が良くないので、1 つの関数の中にたくさんの処理を詰め込んでしまうと、その関数の中で何をやっているのかを理解することが困難になります。ここでは、大域変数は、プログラム中のどこからでもアクセスできるということに着目して、プログラムを書き換えてみます。サンプル 7-19 の draw 関数の中では、

デカルトの「検討しようとする難問をよりよく理解するために、多数の小部分に分割すること」という考え方が基礎にあります。

1. 背景を白色にする
2. ボールを移動させる
3. ボールの壁での反射処理を行う
4. ボールを描く

ということを行っています。そこで、処理 2,3,4 を独立した move, bounce, display 関数として定義することにします。また、中心座標と半径を指定して円を描く関数 drawCircle を定義します。このよう

な方針で書き換えを行ったものがサンプル 7-20 です。

移動するボールその 2（関数化版）サンプル 7-19

```
int xPos;
int speed;
int radius;
oid drawCircle(float x, float y, float r) {
  ellipse(x, y, 2*r, 2*r);
}
void display() {
  stroke(0);
  fill(127);
  drawCircle(xPos, height/2, radius);
}
void move() {
  xPos += speed;
}

void bounce() {
  if ((xPos+radius) > width) {
    speed = -1;
    xPos = width-radius;
  }
  else if ((xPos-radius) < 0) {
    speed = 1;
    xPos = radius;
  }
}

void setup() {
  size(400, 200);
  smooth();
  xPos = width/2;
  speed = -1;
  radius = 20;
}

void draw() {
  background(255);
  display();
  move();
  bounce();
}
```

このように書き換えると、ここの処理が独立して書かれることになるので、1 つ 1 つの処理がやっている内容が明確になると思います。

自分で定義した関数は、自由に使うことが出来ます。つまり、自分で定義した関数の中で、自分の定義した関数を利用することが出来ます。サンプル 7-20 は、自分で定義した関数を自分で定義した関

モジュール化（関数の利用）の特徴として、「複雑な機能を単純な独立した機能に分割して管理する」があります。

数の中で使うものです。ここまで来ると、かなり複雑なプログラムを作れるようになっている筈です。

某アニメキャラもどきを表示 サンプル 7-20

```
// 目を描く
void drawEye(float x, float y, float r) {
  pushMatrix();
  translate(x, y);
  noStroke();
  fill(0, 80, 55);
  ellipse(0, 0, r*2, r*2);
  fill(0, 80, 40);
  ellipse(0, 0, r*2*0.5, r*2*0.5);
  rotate(-PI/4);
  translate(r*0.7, 0);
  fill(0, 0, 99);
  ellipse(0, 0, r*2.0*0.3, r*2.0*0.3);
  popMatrix();
}
//  口を描く
void drawMouth(float x, float y, float w, float h) {
  pushMatrix();
  translate(x, y);
  noFill();
  stroke(0, 0, 0);
  bezier(-w, 0, -w, h, 0, h, 0, 0);
  bezier(w, 0, w, h, 0, h, 0, 0);
  popMatrix();
}
// 顔全体を描く
void drawQB(float x,float y,float w,float h){
  drawEye(x-w/2,y,30);
  drawEye(x+w/2,y,30);
  drawMouth(x,y+0.4*h,35,20);
}
void setup() {
  size(400, 400);
  colorMode(HSB, 359, 99, 99);
  smooth();
}
void draw() {
  background(0, 0, 99);
  drawQB(mouseX,mouseY,width/2,height/2);
}
```

情報メディア基盤ユニットの単位は必ず取得してよ。必修科目の単位を落としていると卒研につけないんだ。卒研をクリアできないと卒業できないんだ。これは契約だよ。

顔の輪郭部分なども欲しい気がするのですが。そうすると耳とかもいるのかな？
でも、シンプルな方が良いかな？