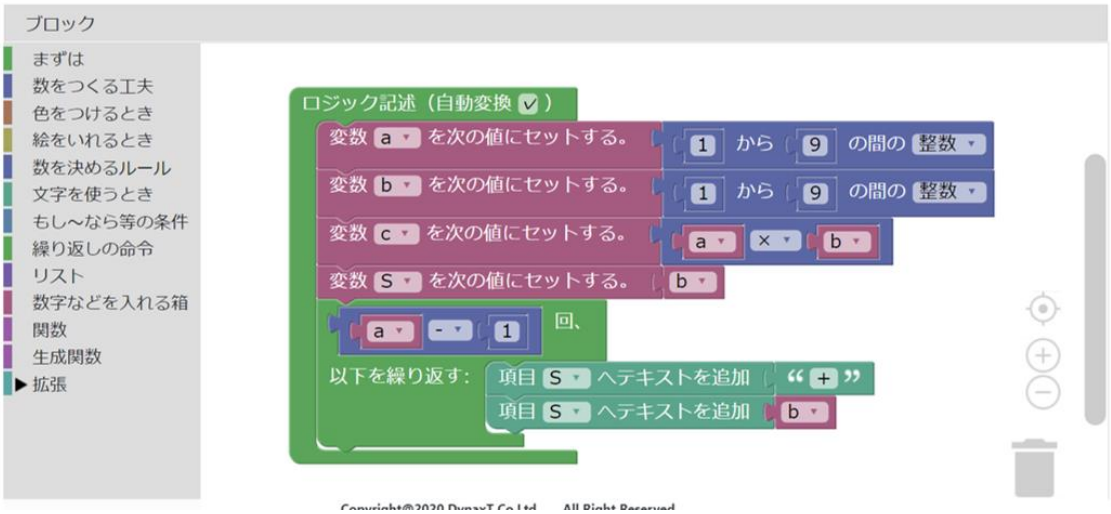
Blockly（ぶろっくりー） の つかいかた

1. Blocklyとは

Blocklyとは、Googleが提供するビジュアルプログラミング言語です。
MathPubでは、このBlocklyを使用してプログラミングをおこないます。



MathPubで使っているBlocklyは、Google Blocklyをさんすうの問題づくりにあわせてアレンジしたものです。

ブロックの動きや機能についてお問い合わせはMathPubセンターへお寄せください。

プログラミングをはじめるにあたって

コンピュータは同じ作業を繰り返すことや計算問題を解くことは得意ですが、
あいまいな指示を理解したり、相手の思いや考えを想像するのは苦手です。

また、コンピュータはプログラムされたとおりにしか動きません。

なので、プログラミングをするときは
はっきりとわかりやすい、誤解のないような表現をするようにしましょう。

2. 画面の説明

まず、Blocklyの画面について説明します。

Blocklyメニュー

それぞれの項目をクリックすると、ブロックのひきだしが開きます。

[さぎょうエリア] ・・・ このなかでブロックを組み立てます。

[スクロールバー] ・・・ 表示する位置を動かせます。

[きほんにもどす] ・・・ 画面内におさめられるサイズに戻します。

[おおきくする] ・・・ ブロックの表示サイズを大きくします。

[ちいさくする] ・・・ ブロックの表示サイズを小さくします。

[ごみばこ]

ブロックをさぎょうエリアから消したいとき、
ブロックをドラッグしながらごみばこにむかって動かすと、
ブロックを捨てることが出来ます。

3. 基本的なつかいかたと動き

教材を作成するとき、先生などから出題された教材に回答するとき…
そんなときは教材編集画面を操作します。

教材編集画面は以下の構成になっています。

①教材情報

教材のタイトルや算数の単元としての学習する学年、検索のときに便利なキーワードなどを設定します。

②問題

難易度、問題タイトル、問題文の意図、問題文、画像登録の項目があります。

問題タイトルは「もんだいをときましょう」などの指示を書きます。

問題文の意図は、問題文を作成するときに気を付けること、条件や決まりごとがあれば書きます。

問題文は解いてもらいたい計算式や文章問題を書きます。

③解答

難易度、解答タイトル、解答文の意図、解答文、解答の項目があります。

解答文の意図は、解答文を作成するときに気を付けること、条件や決まりごとがあれば書きます。

解答文は問題に対応した解き方・考え方や答えを書きます。

解答は、答えのみを書きます。

④プログラム

難易度、プログラムタイトル、プログラミング言語区分、プログラムの意図、プログラムの項目があります。

プログラムの意図は、プログラムを作成するときに気を付けること、条件や決まりごとがあれば書きます。

ここからは

「じぶんできめたかず と ランダムにつくられたかずのたしざん」

という教材をつくることを例に、つかいかたと動きを説明していきます。

☆もともとMathPubに登録・公開されている「きほんのつかいかた」という教材を

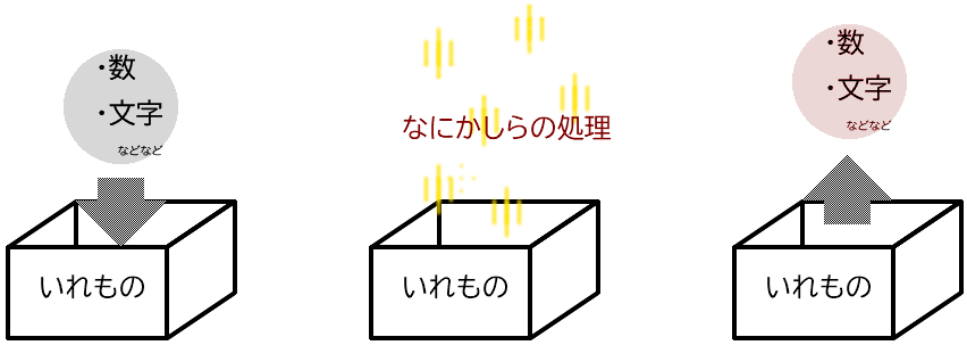
コピー → 保存 して使うとBlockly部分以外を設定しなくてすむのでらくちんです。

メインメニューの「教材の検索・問題を解く」で教材検索画面を開き、
タイトルに「きほんのつかいかた」で検索してください。

プログラミングでは、かずやもじをしまっておく“いれもの”が必要になります。
かずをそのまま扱うのではなく、

かずをいれものに入れる→処理をする→いれものから取り出して表示する

というように扱います。



このいれもののことを**「変数」**とよびます。

「変数」といわれたら、「数や文字をいれるいれもののことだ！」と覚えておいてください。

問題文や解答文など、Blocklyの作業エリアの外からは、この変数につけた名前で呼び出します。
例として、「きほんのつかいかた」という教材の「問題文」と「解答文」は次のように書かれています。

【問題文】

\(じぶんできめたかずは{A}です。 \)
\(ランダムにつくられたかずは{B}です。 \)
\(この２つのかずをたすといくつですか？ \)

【解答文】

\(じぶんできめたかずは{A}です。 \)
\(ランダムにつくられたかずは{B}です。 \)
\(この２つのかずをたすと{C}です。 \)

これは、A という変数と B という変数と C という変数の 3 つの変数を使っています。

それぞれ、
・A にはじぶんできめたかずをしまう
・B にはランダムにつくられたかずをしまう
・C には、A と B をたしあわせた答えをしまう

というルールになっているので、このようになります。
このルールは教材を作る人が決めるのですが、じぶんで問題文や解答文を作るときは、A や B や C でなくても、じぶんの好きな名前にすることが出来ます。

※問題文や解答文で変数を呼び出すときは、{}(ちゅうかっこ)でくくります。
※MathPubで使う変数名は英数字だけでなく、日本語でもよいです。

MathPubでは、さまざまなブロックをつなげてプログラミングを行います。
そのかたまりのことをプログラムとよびます。
プログラムというのは、コンピュータに「こんなお仕事をしてね！」とお願いする命令みたいなものです。
このお仕事ひとつひとつを処理とよびます。

基本的にコンピュータは、上から順番に処理をしていきます。

例　バタートーストが食べたいとき

食パンを 1枚用意する。
↓
食パンをトースターにいれる。
↓
タイマーを 3分にセットする。
↓
焼きあがった食パンを取り出す。
↓
食パンにバターをぬる。

もし焼く前にバターを塗りたいときは

食パンを 1枚用意する。
↓
食パンにバターをぬる。
↓
食パンをトースターにいれる。
↓
タイマーを 3分にセットする。
↓
焼きあがった食パンを取り出す。

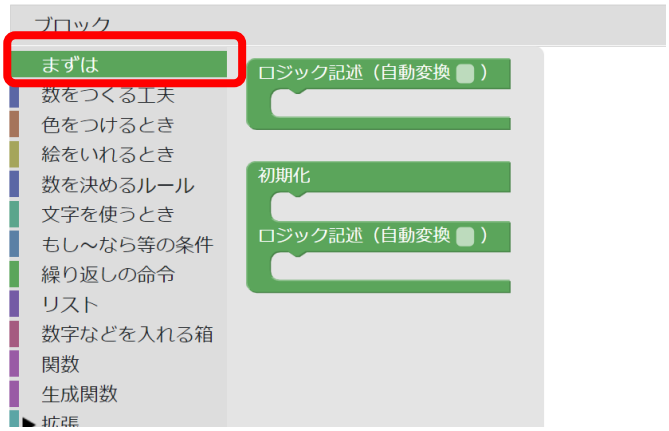
となると思います。

このように、結果を想像しながら、処理の順番も考えるようにしましょう。

4. 各ブロックの機能と使用例

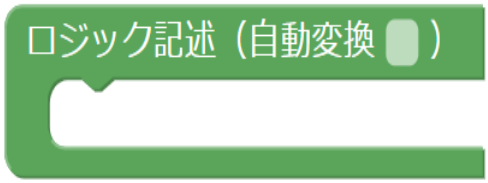
1. ロジック記述ブロック

プログラムのブロックはこのロジック記述ブロックの中につなげます。
このロジック記述ブロックにつながっていないブロックは、プログラムの中に組み込まれません。



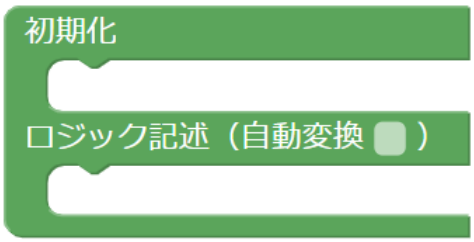
ロジック記述ブロックは、2種類あります。

＜ロジック記述ブロック＞



基本的には、こちらのブロックを使います。
自動変換□ については、
＜絵を入れるブロック＞を使うときはチェックしないようにしてください。
基本的には**チェック**✓をいれておきましょう。

＜初期化・ロジック記述ブロック＞



＜ロジック記述ブロック＞の代わりに、
この**＜初期化・ロジック記述ブロック＞**を使うこともできます。
変数ブロックなどで問題生成時に初期化しておきたいブロックを初期化部分につなげます。

2.　変数ブロック(かずのいれもの)を用意しよう

問題文や解答文に合わせて**変数ブロック**を用意します。

もう 1度、「きほんのつかいかた」という教材の「問題文」と「解答文」を見てみます。

【問題文】

\(じぶんできめたかずは{A}です。 \)
\(ランダムにつくられたかずは{B}です。 \)
\(この 2 つのかずをたすといくつですか？ \)

【解答文】

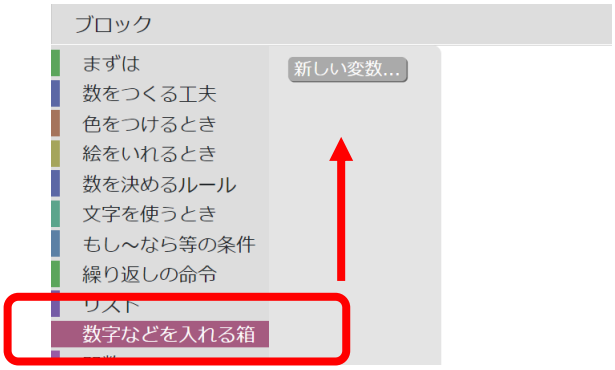
\(じぶんできめたかずは{A}です。 \)
\(ランダムにつくられたかずは{B}です。 \)
\(この 2 つのかずをたすと{C}です。 \)

これは、A という変数と B という変数と C という変数の 3 つの変数を使っていますので、その分のブロックを用意してあげましょう。

変数は自分の好きな名前を付けることが出来ました。
なので、最初から用意されているわけではありませんから、自分で作らないといけません。

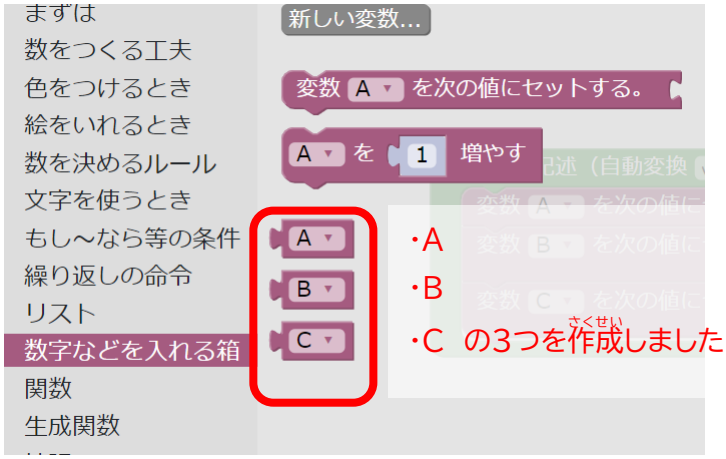
順番に見ていきましょう。

① 「数字などを入れる箱」を選択すると引き出しが出てきますので、
そのなかにある「新しい変数…」をクリックしましょう。



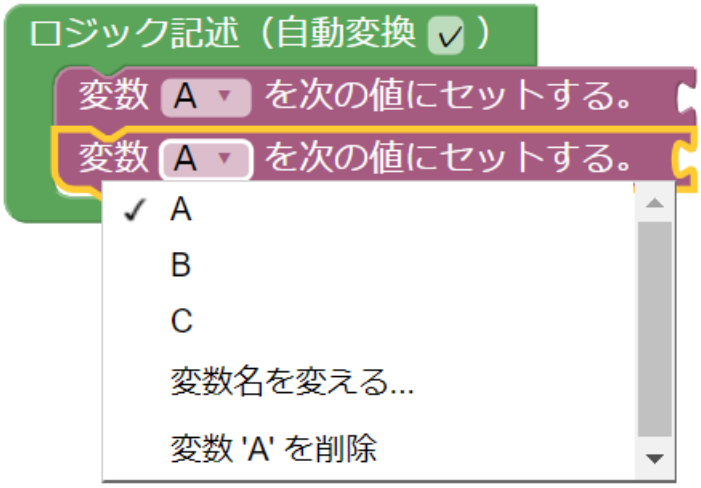
② すると、こういったメッセージと入力エリアが表示されますので、作りたい変数の名前を入力します。

今回は A という変数と B という変数と C という変数の 3 つの変数を作ります。



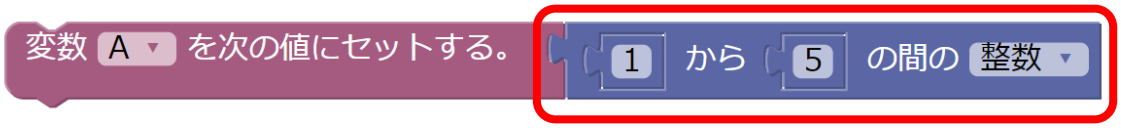
これで、<**変数ブロック**>を作ることができました。

使うときは、このように変数名を切り替えて使います。



同じ引き出しには、このように<**変数の中身の数を増やすブロック**>もあります。
決まった数を増やしたいときに便利です。

数をつくるブロックなどをつなげて使う。



この<**変数ブロック**>には、次に紹介する**数をつくるブロックなど**を使って中身をセットします。その中身のことを**値**とよびますので、覚えておきましょう。

3.　数をつくろう

さて、先ほどの＜**変数ブロック**＞に値をいれます。使うのは、**数をつくるブロック**です。

＜**決まった数をつくるブロック(定数ブロック)**＞です。
好きな数を入力することが出来ます(小数も入力出来ます)。

＜**ランダムな数をつくるブロック(乱数ブロック)**＞です。

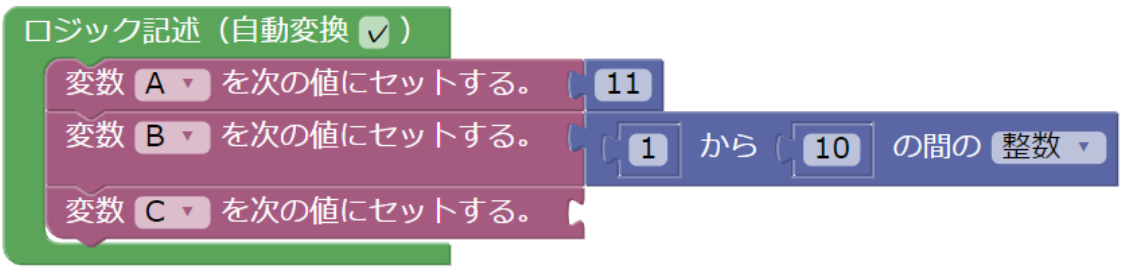
ランダムな数というのは、コンピュータが勝手に作ってくれる数です。
サイコロを振る、バラバラになるようによく切ったトランプの山から 1枚引く…みたいなことをコンピュータがやってくれます。

「1」から「10」の間の…とあるように、一番小さな数と一番大きな数を決めてあげると、その範囲の中でランダムな数を作ってくれます。
この場合は 1,2,3,4,5,6,7,8,9,10 の中のどれかになります。

「きほんのつかいかた」という教材の **A という変数と B という変数と C という変数の 3 つの変数** について、さっそくこのブロックを使って値を入れてみましょう。

・A にはじぶんできめたかずをしまう
・B にはランダムにつくられたかずをしまう
・C には、A と B をたしあわせた答えをしまう
　・・・ という決まりとなっていますので(教材を作ったひとがそう決めました)、

・A には**決まった数をつくるブロック(定数ブロック)**
・B には**ランダムな数をつくるブロック(乱数ブロック)**　　をつなげてあげましょう。



つなげてみました。例として、**A のじぶんできめたかず**は**11**にしています。
みなさんはどんな数字にしますか？好きな数を入れてみましょう。

さて、C にはどちらのブロックをつなげればいいのでしょう？

C につなげるのは、**決まった数をつくるブロック(定数ブロック)**でも**ランダムな数をつくるブロック(乱数ブロック)**でもありません！

・C には、A と B をたしあわせた答えをしまう
となっていますので、このあと紹介する**＜計算ブロック＞**をつなげます。

※さまざまな＜ランダムな数をつくるブロック＞テクニック

＜ランダムな数をつくるブロック＞は決められた範囲内の整数をつくるブロックですが、
このあと紹介する、ほかのブロックと組み合わせることで整数以外も作ることが出来ます。

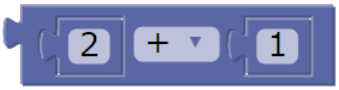
小数を作りたいときは、「10」から「20」の間の…としてあげて、この後出てくる
＜計算ブロック＞を使って、「ランダムな数÷10」としてあげると、
1.0～2.0 の間の小数を、0.1 きざみで作ることが出来ます(→4. 計算をしてみよう)。

分数を作りたいときは、**分子用の数**と**分母用の数**をそれぞれ作って、
＜分数をつくるブロック＞を使うことで作ることが出来ます(→4. 計算をしてみよう)。

これらのブロックについては、またそれぞれのページで詳しく説明します。

4.　　計算をしてみよう

ここでは＜**計算ブロック**＞について紹介します。



＜**たしざんブロック**＞
この場合は 2+1 の計算を表していて、値は 3 となります。

＜**ひきざんブロック**＞
この場合は 2-1 の計算をしていて、値は 1 となります。

＜**かけざんブロック**＞
この場合は 1×2 の計算をしていて、値は 2 となります。

＜**るいじょう(累乗)ブロック**＞
この場合は 2を 2回かけざんしているのと同じで(2×2)、
値は 4 となります。

※るいじょう(累乗)とは

るいじょう(累乗)とは、同じ数をなんどもかけていく計算のことです。
式としては 2^2 のように書き、**2の2じょう**と読みます。
これをコンピュータで表すときは **^(ハット、もしくはキャレットとよびます)** という記号を使って、2^2 と書きます。

$$2^2 = 2\text{^}2 = 2\times2 = 4$$

ほかの数でも同じように計算します。

$$2^8\text{(2の8じょう)} = 2\text{^}8 = 2\times2\times2\times2\times2\times2\times2\times2 = 256$$
$$3^4\text{(3の4じょう)} = 3\text{^}4 = 3\times3\times3\times3 = 81$$

これは中学校の数学で学習する内容ですので、

まだ習っていないよ！という方は、先生に聞いてみてください。

＜わりざんブロック＞
この場合は　2÷1 の計算をしていて、値は 2.0 となります。

＜わりざんブロック＞を使うと、割り切れて答えが整数(小数点以下が0)となっても、
2. 0というように小数の形で表示されます。
これを整数のみの表示にしたい(2 と表示させたい)場合は、＜四捨五入・切り上げ・切り捨てブロック＞を使います(→5. いろんな計算や数えるためのブロック)。

＜分数をつくるブロック＞

まえの数字が分子、うしろの数字が分母です。
この場合は $\frac{1}{3}$ となります。

値としては$\frac{1}{3}$ではなく、1÷3 という式が入っている状態です。
きれいな分数の形で表示するためには、分数表示のためのコマンドが必要です。

※「問題文」や「解答文」での分数の表示について

＜分数をつくるブロック＞の説明でも書きましたが、このブロックをつなげた変数を
そのまま表示させると、「1÷3」というような式が表示されます。

そのままだと使いづらいので
「問題文」や「解答文」では次のように書いてあげます。

\frac{{{分子}}}{{{分母}}}

たとえば、X という変数に分子、Y という変数に分母 となるような値が入っていれば、

\frac{{{X}}}{{{Y}}}

と書くことで、分数の形で表示されます。

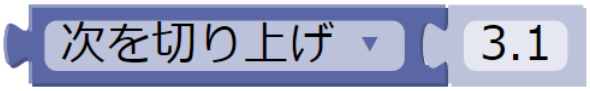
5.　　いろんな計算や数えるためのブロック

そのほかにも値を求めたり、値を整えたりするためのブロックがあります。

＜四捨五入ブロック＞

後ろにつなげたブロックの値を小数第一位で四捨五入した整数にします。
この場合は 3.1 → 3 となります。

＜切り上げブロック＞



後ろにつなげたブロックの値を小数第一位で切り上げした整数にします。
この場合は 3.1 → 4 となります。

＜切り捨てブロック＞

後ろにつなげたブロックの値を小数第一位で切り捨てした整数にします。
この場合は 3.1 → 3 となります。

また、これらのブロックは**＜わりざんブロック＞**などによって小数の形になってしまった値を整数にするためにも使えます。 例:36.0 → 36

＜余りをもとめるブロック＞

わりざんの商ではなく、あまりだけをもとめたいときに使います。この場合は 64÷10 のあまりなので 4 となります。

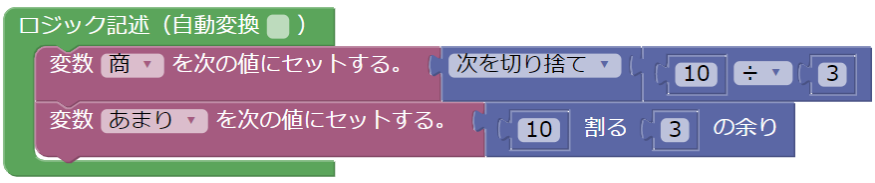
＜わりざんブロック＞は割り切れない場合、小数の形で答えをだします。

例:10÷3 ＝ 3.3333333333…

これでは商はわかってもあまりがわかりません。
そこで、**＜余りをもとめるブロック＞**を使います。

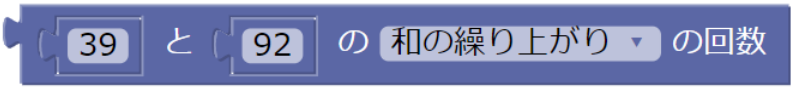
例:10÷3 のあまり ＝ 1

このふたつのブロック、さらに**＜切り捨てブロック＞**を合わせてつかうことで、商とあまりを別々に求めることが出来ます。

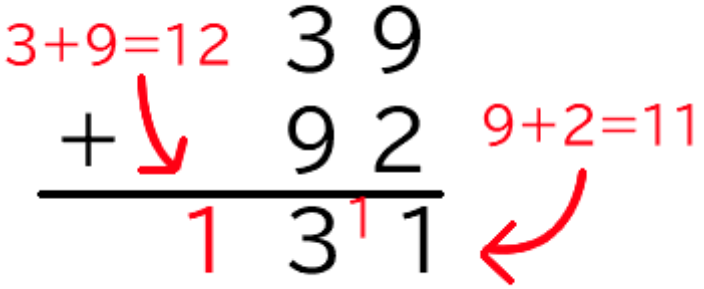


これで、**商という変数**には「3」、**あまりという変数**には「1」が入ります。

＜繰り上がりの回数ブロック＞



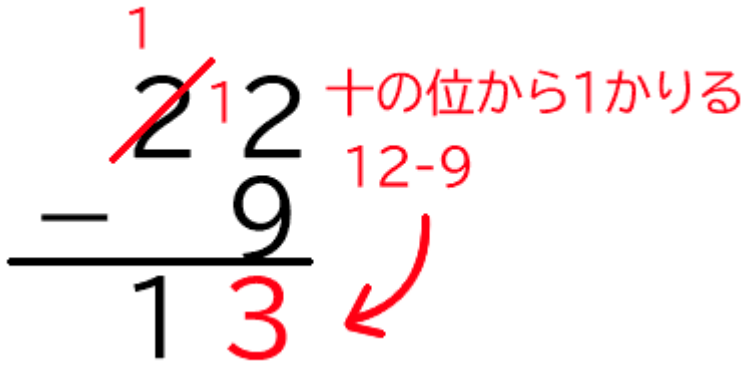
2 つの数をたしざんしたとき、何回繰り上がりがあるかを数えてくれます。
この場合だと 39 + 92 を計算するのですが、



一の位の 9+2=11 で **1回**、十の位の 3+9=12 で **1回**と繰り上がりの回数を数え、
合計**2回**繰り上がりがあるということになります。 なので、値は 2 となります。

＜繰り下がりの回数ブロック＞

2 つの数をひきざんしたとき、何回繰り下がりがあるかを数えてくれます。
この場合だと 22 - 9 を計算するのですが、

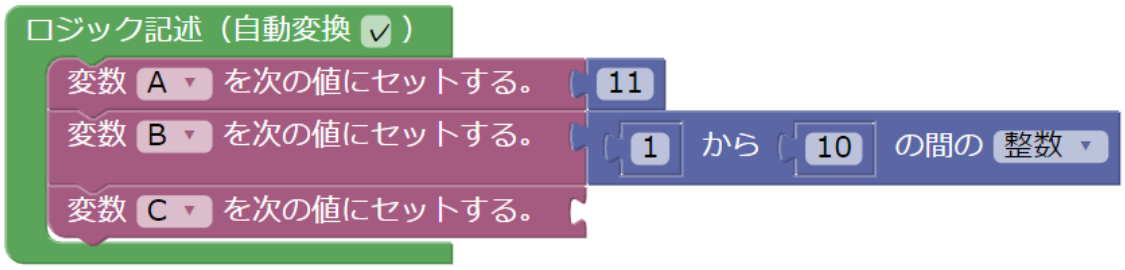


「十の位（となりの位）から 1 かりる」ということを 1回行うので、合計**1回**繰り下がりがあるということになります。 なので、値は 1 となります。

さて、また「きほんのつかいかた」という教材のプログラミングの続きです！

ここまで、**A という変数**と **B という変数**と **C という変数**の **3 つの変数** のうち、
変数A と **変数B** については、次のようなブロックをつなげました。

・A には**決まった数をつくるブロック（定数ブロック）**
・B には**ランダムな数をつくるブロック（乱数ブロック）**

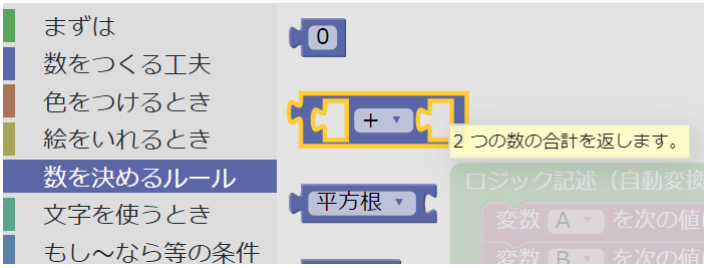


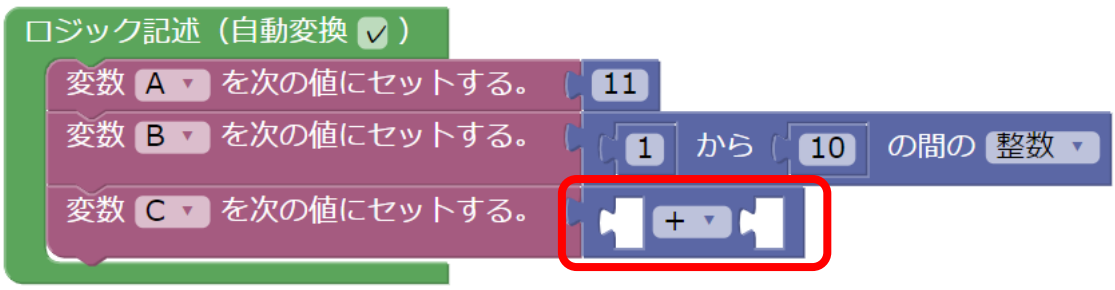
＜11＞となっているところは、みなさんそれぞれ違うと思いますが、こんな感じのブロックになっているのではないでしょうか？

さぁ、いよいよ**変数C** にもブロックをつなげてみましょう。

たしか、

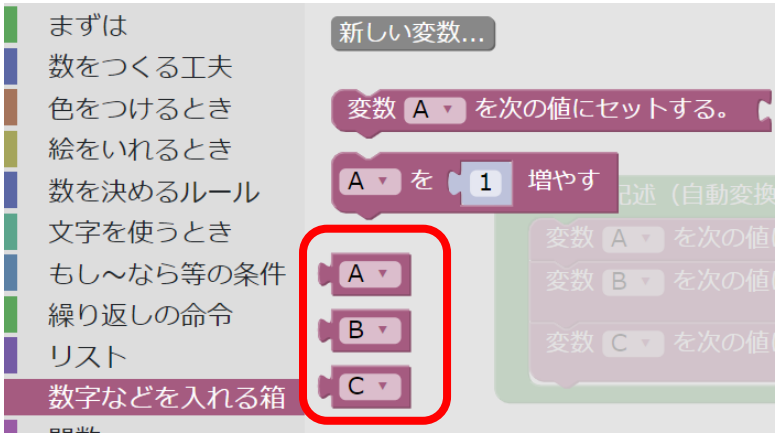
・C には、A と B をたしあわせた答えをしまう
・・・ という決まりとなっていますので（教材を作ったひとがそう決めました）、
先ほど紹介した＜**たしざんブロック**＞をつなげてあげるとよさそうです。



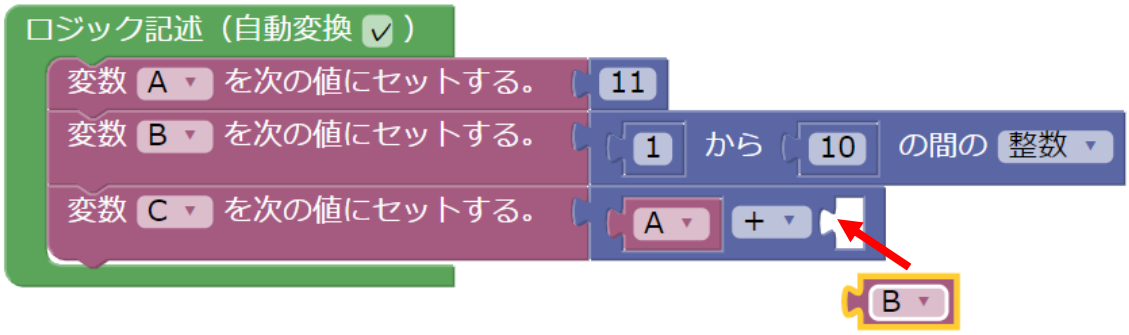


つなげました！

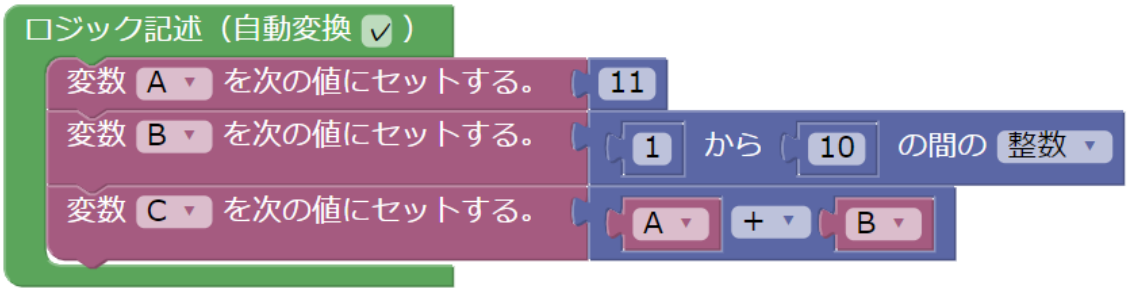
でも、このままだと何と何をたしざんしたらいいのか、コンピュータはわかりません。
なので、<**変数ブロック A**>と<**変数ブロック B**>をはめ込んであげましょう。



この変数ブロックを…



カチッとはめこみます！



はい！　これで、「きほんのつかいかた」という教材の**プログラムが完成しました！**

ブロックエリアのすぐ左上に**問題生成ボタン** 問題生成 がありますので、ぜひ押してみてください。どうでしょうか？

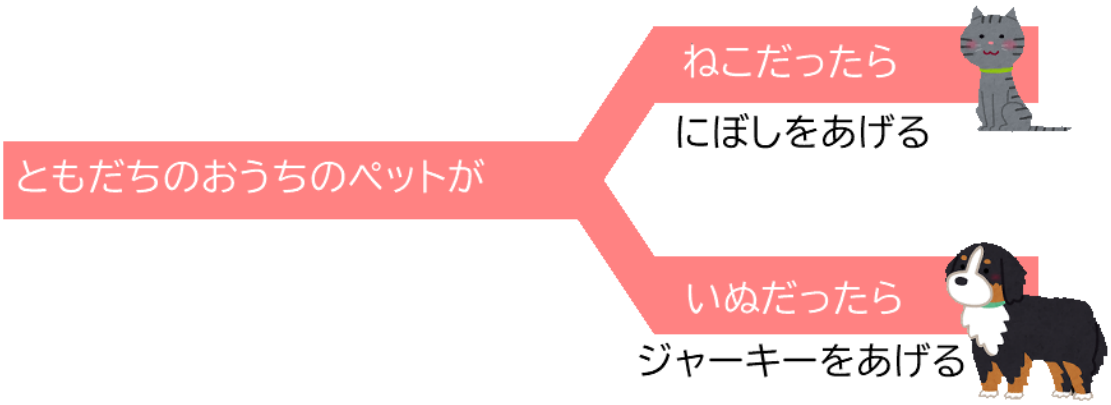
> ※「エラーが発生しました」というメッセージが表示された場合は、
> ブロックの組み方が違っていないかを確認してみてください。

6. 条件によって処理を変えたい

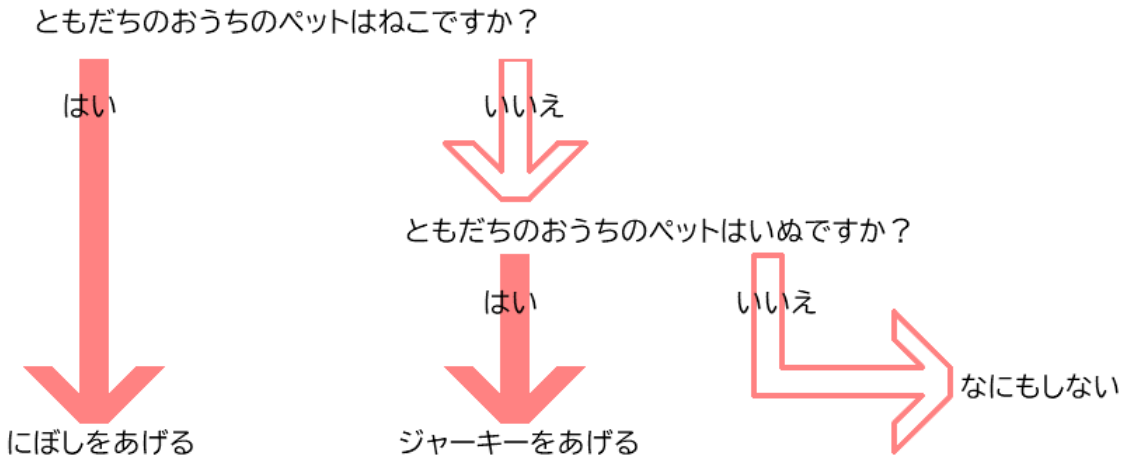
コンピュータに「こんなお仕事をしてね！」とお願いする命令みたいなもの、それがプログラミングなのですが、

> 「友達のおうちのペットが犬の場合はジャーキーをあげて、猫の場合はにぼしをあげてね！」

というように、場合・条件によってお願いしたい処理を変えてもらうことが出来ます。



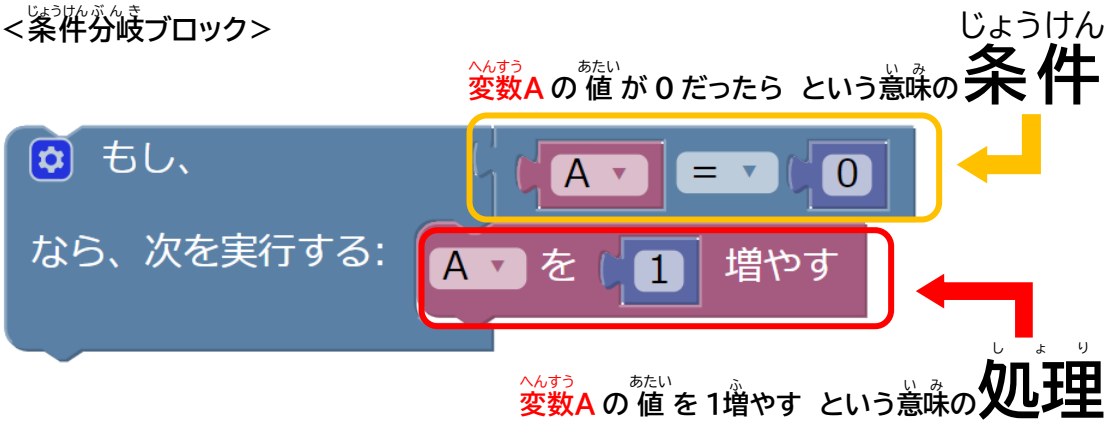
こういったお願いするには、基本的に はい / いいえ で判断し、それによって処理を分けるようにします。



こんなかんじで考えるとよいです。

こんなときは、これから紹介する＜**条件分岐ブロック**＞を使います。

＜条件分岐ブロック＞



これは、

変数A の値は 0 ですか？「はい」なら、変数A の値 を 1増やす処理をしてください。

というお願いをあらわしています。

「**変数A の値 が 0**」 が条件です。ここは、**＜判定ブロック＞**をつなげます。

この条件にあてはまったときの処理として

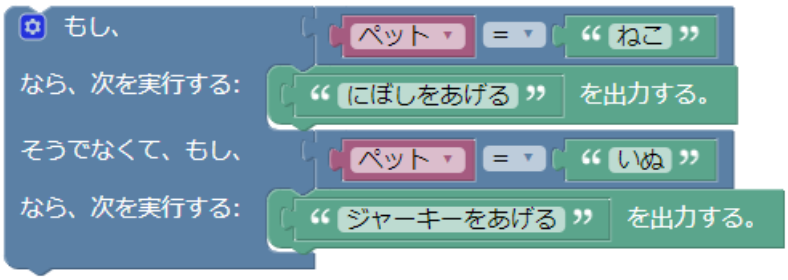
＜変数の中身の数を増やすブロック＞ 

をつなげています。

ブロックの左側がまっすぐになっているブロックなら、ブロックをつなげることが出来ます（**＜ロジック記述ブロック＞**はつなげられません）。 個数に制限もないので、処理に合わせてつなげてください。

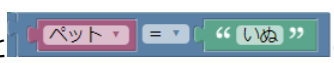
＜条件分岐ブロック＞は左上の  から、条件と処理のセットを増やすことが出来ます。

おともだちのおうちのペットの例をブロックで表すと…



こんな形になります。

これは 

と 

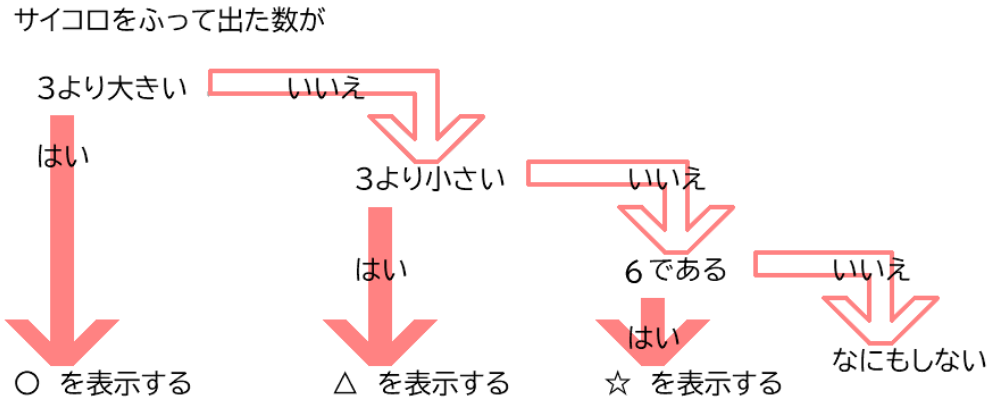
の順番が逆でも同じなので

どちらでもよいのですが、次のような場合はよく考えてあげないと、プログラムが複雑になったり、コンピュータが上手に判断できなくなったりします。

例)サイコロをふって出た数によって表示されるマークを変えたい！

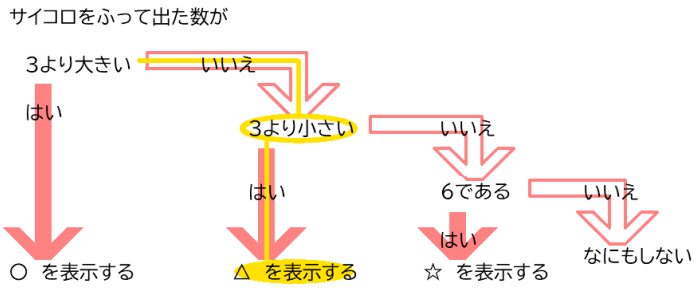
① 3 より大きい数だったら「○」を表示する。
② 3 より小さい数だったら「△」を表示する。
③ 6 だったら「☆」も表示する。

さて、どのように考えてあげたらよいでしょうか？
先ほどと同じように、①から順番に「はい」と「いいえ」で分岐する図にしてみます。

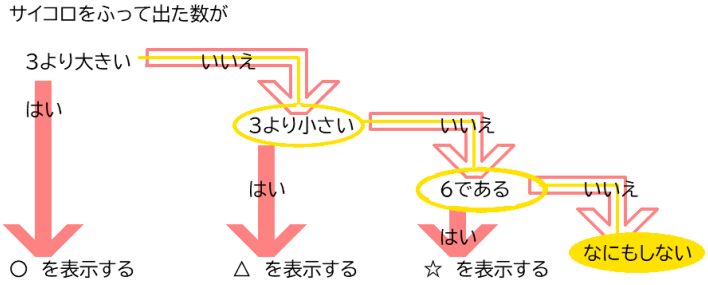


さて、これで思った通りに処理をしてくれるのでしょうか？
プログラミングの前に、サイコロの目によってどんな処理結果になるのかを考えてみましょう。

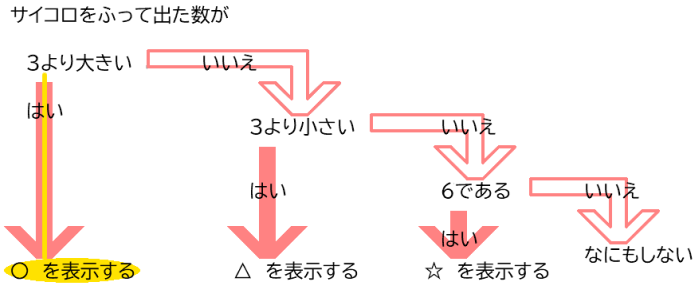
・1 がでたとき と 2 がでたとき



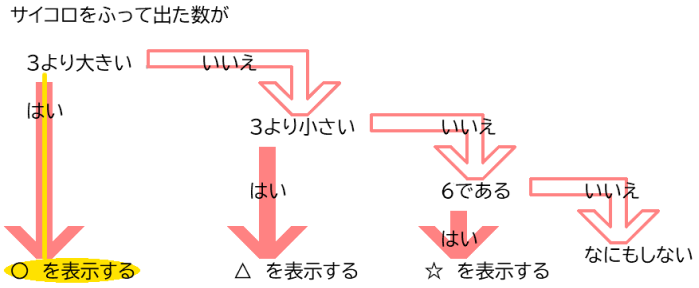
・3 がでたとき



4 がでたとき と 5 がでたとき



6 がでたとき

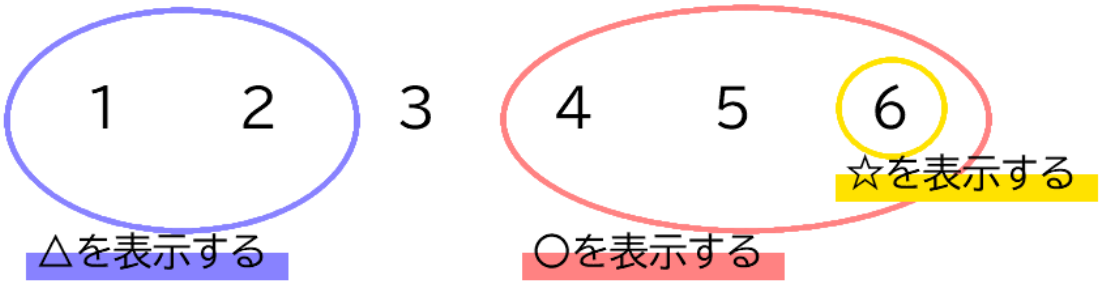


・・・・・・・・・あれ？？？？？(;’∀’)

6がでたときは☆も表示してもらいたいのに、
①～③の順番で処理をすると〇だけしか表示されません！！

このように、「はい」と「いいえ」で順番に判断していくようなプログラムでは、
どの順番で条件を組み立てるかがカギとなっています。

さて、あらためて考え直してみます。そもそも、条件はこのままでいいのでしょうか？
まずは、グループわけの図を描いてみましょう。



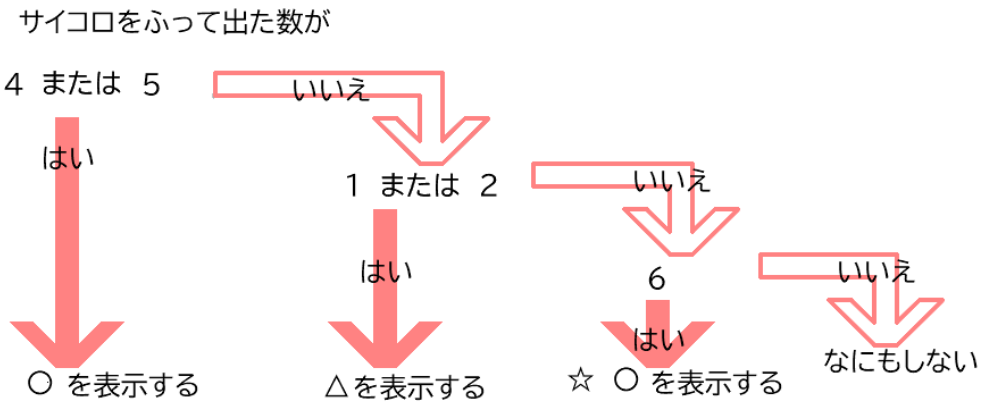
どんな処理がしたいか、でグループわけしました。

この図をみてみると、最初に書かれた①～③の条件がわかりにくいです。
シンプルになるように考えてみましょう。

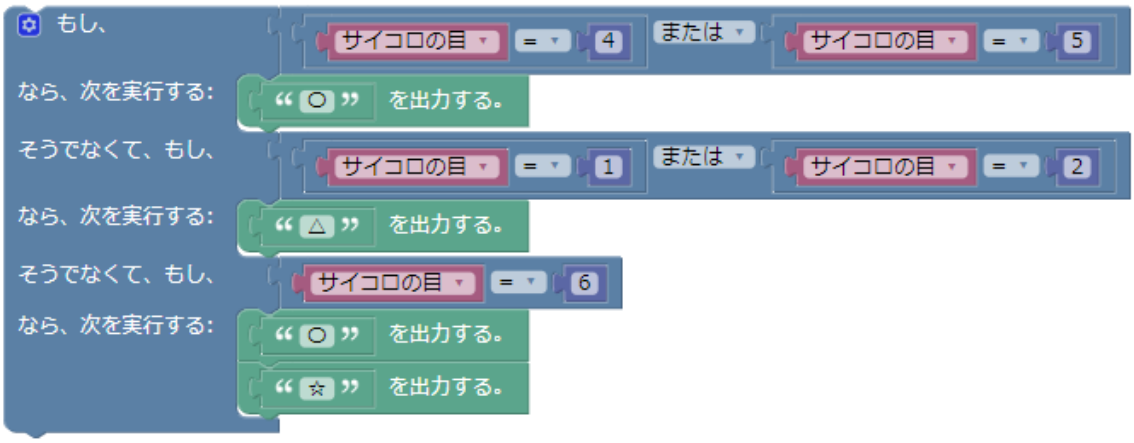
例)サイコロをふって出た数によって表示されるマークを変えたい！

① 3 より大きい数だったら「〇」を表示する。 → 4,5 だったら「〇」を表示する。
② 3 より小さい数だったら「△」を表示する。 → 1,2 だったら「△」を表示する。
③ 6 だったら「☆」も表示する。 → 6 だったら「〇」と「☆」を表示する。

どうでしょうか？ これならわかりやすくなったと思いませんか？
また「はい」と「いいえ」で分岐する図にしてみます。



こうなりました！



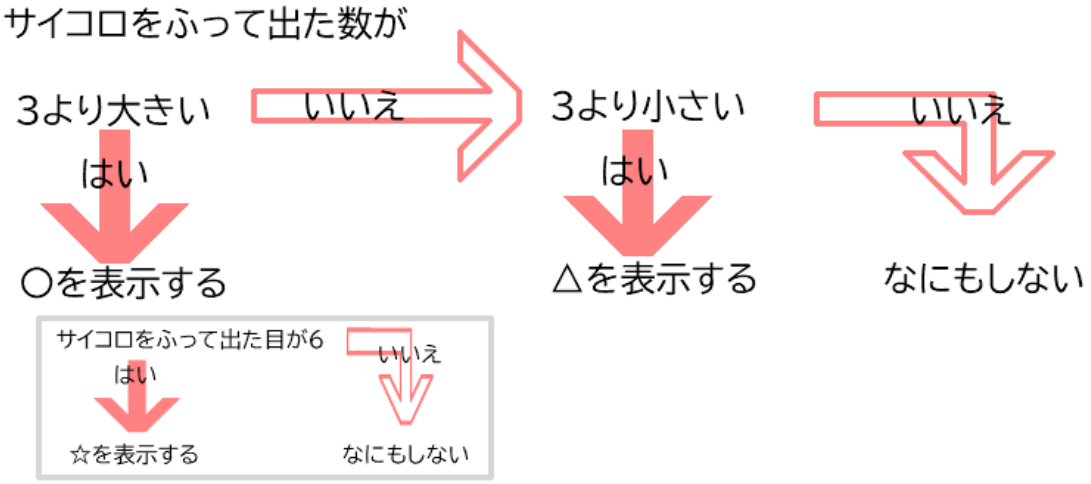
ブロックを組み立てるとこんな感じです。

考え方のヒントとしては、いちばん小さい(中身がすくない)グループから順番に条件を組み立てるとよいと思います。

＜**条件分岐ブロック**＞は、処理部分に＜**条件分岐ブロック**＞をつなげることができます。
なので、

① 3 より大きい数だったら「○」を表示する。
→ さらに 6 だったら「☆」を表示する。
② 3 より小さい数だったら「△」を表示する。

というように、①の処理のなかに条件分岐を入れ込むことが出来ます。すると…



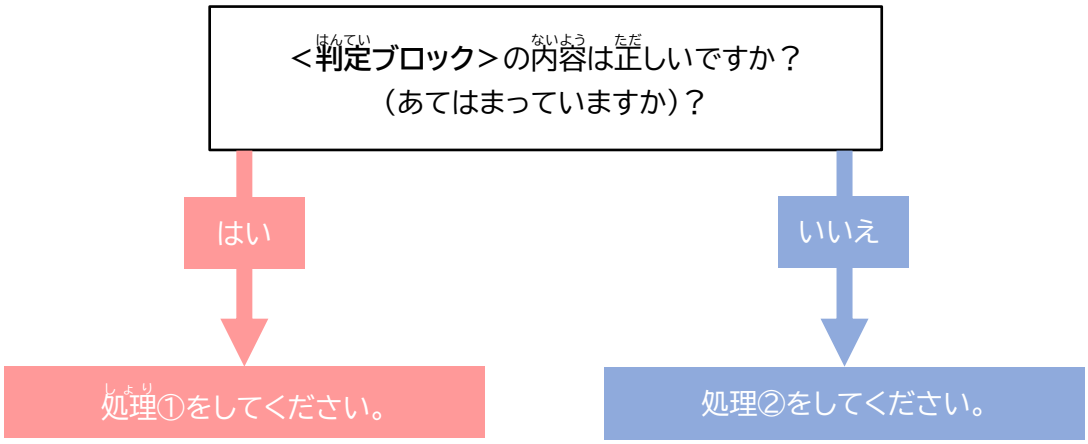
6 のときは、○を表示したうえで、さらに☆を表示する。

というようなプログラムとなり、先ほどのブロックと同じ結果となります。

このような形をしたものを、入れ子構造とよびます。
ロシアのおもちゃであるマトリョシカみたいに、人形の中に人形が入っていて、さらにその中に人形が入っている…みたいなつくりをイメージしてください。

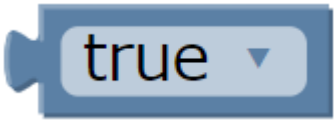
7.　これって〇〇ですか？＜判定ブロック＞

コンピュータは基本的に はい / いいえ で判断しますよ、という話をしました。
先ほどの＜**条件分岐ブロック**＞では何を判断しているかというと、
「＜**判定ブロック**＞の内容は正しいですか（あてはまっていますか）？」という質問に対して
はい / いいえ で判断しています。



処理については、「なにもしない（ブロックをおかない）」ということもできます。

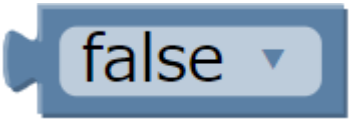
この＜**判定ブロック**＞について、紹介していきます。



＜**trueブロック**＞
このブロックがつながっていると、コンピュータは無条件で
「はい」「あてはまっている」と判断します。

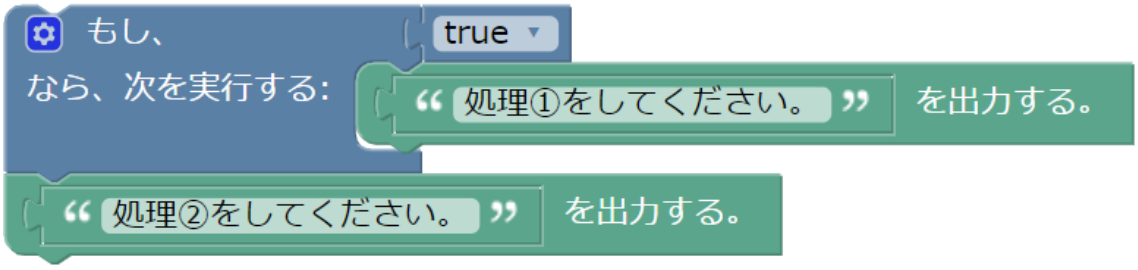
どんなときに使うかというと、「ぜったいにこの処理はしてほしい！」というときに使います。

このあとでてくる＜**くりかえしブロック**＞と組み合わせて使うことが多いかもしれません。
true は日本語で **本当の、正しい、真の、あてはまる** という意味の英語です。



＜**falseブロック**＞
このブロックがつながっていると、コンピュータは無条件で
「いいえ」「あてはまっていない」と判断します。
false は日本語で **事実と異なる、偽の、うその** という意味の英語です。

このようなブロックを組んだとき、コンピュータはどんな処理をするのでしょうか？



この場合、条件の部分には＜**trueブロック**＞がつながっています。
つまり、絶対に**「処理①をしてください。を出力する」**ということになります。

そして、＜**条件分岐ブロック**＞のすぐ次のブロックである、
「処理②をしてください。を出力する」というブロックに移ります。

なので、

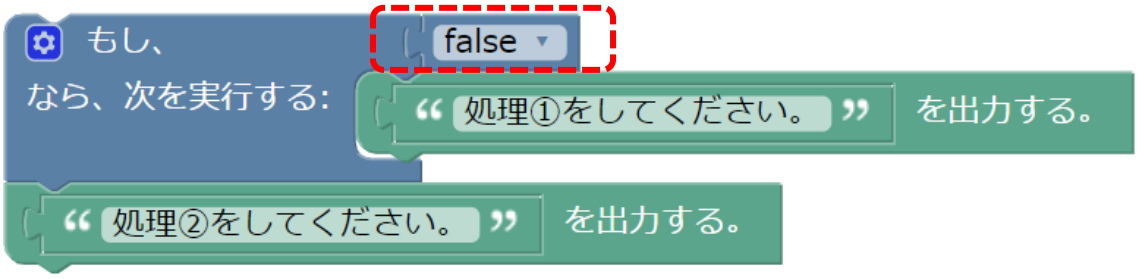
処理①をしてください。 処理②をしてください。

という結果が表示されます。

※＜**出力ブロック**＞について

ちょこちょこ出現している、このブロックですが
これははめ込んだブロックの内容を出力（画面に表示したり、書き出したり）する…
という機能のブロックです。

では、このようにブロックを組むと、コンピュータはどんな処理をするのでしょう？



の部分が、先ほどと変わっています。

この場合、条件の部分には＜**falseブロック**＞がつながっています。
つまり、**「処理①をしてください。を出力する」というブロックは無視する**ことになります。

そして、＜**条件分岐ブロック**＞のすぐ次のブロックである、
「処理②をしてください。を出力する」というブロックに移ります。

なので、

処理②をしてください。

という結果が表示されます。

このように、条件に当てはまらない場合は処理をしないで次のブロックにいきます。

この 2 つの＜**trueブロック**＞と＜**falseブロック**＞は変数ブロックにつなげることもでき、
もっと複雑なプログラムを組むときにとても活躍します。

・おなじ　か　おなじじゃないか　を判定させるブロック

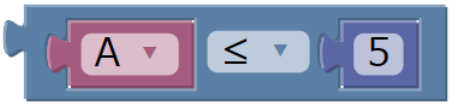
＜＝ブロック＞
変数A の値が 5 と同じ(つまり 5)かどうかを判定します。5 なら「はい」、それ以外は「いいえ」と返します。

＜≠ブロック＞
変数A の値が「5 ではない」かどうかを判定します。
5以外なら「はい」、5なら「いいえ」と返します。

・値のおおきさ　を判定させるブロック

変数A の値が「5 より小さい」かどうかを判定します。
5より小さければ「はい」、大きければ「いいえ」と返します。

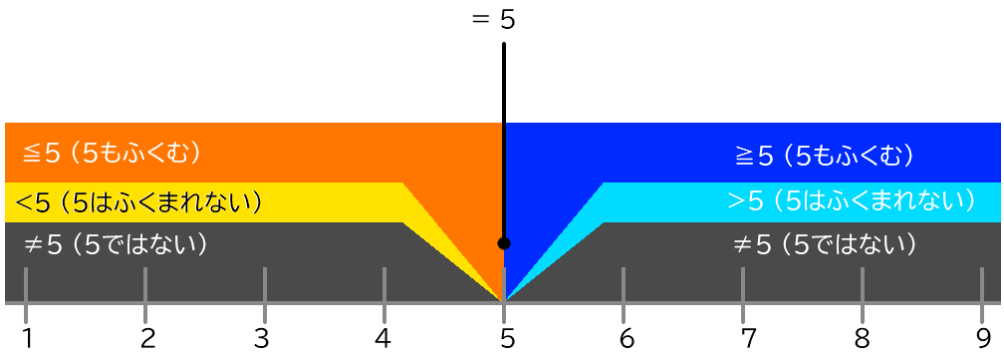
変数A の値が「5 より大きい」かどうかを判定します。
5より大きければ「はい」、小さければ「いいえ」と返します。



変数A の値が「5以下」かどうかを判定します。
5以下なら「はい」、5 より大きければ「いいえ」と返します。

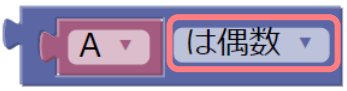
変数A の値が「5以上」かどうかを判定します。
5以上なら「はい」、5 より小さければ「いいえ」と返します。

数の大小関係については数直線で表してみるとわかりやすいです。
基準となる数を含むのか含まないのか、注意しましょう。



・どんな数なのかを判定させるブロック

数の大小だけでなく、どんな数なのかを判定させるブロックもあります。



で囲んだところをクリックすると種類を変えることが出来ます。

変数A の値が「偶数」かどうかを判定します。
偶数であれば「はい」、奇数であれば「いいえ」と返します。

変数A の値が「奇数」かどうかを判定します。
奇数であれば「はい」、偶数であれば「いいえ」と返します。

※偶数・奇数とは

偶数とは、
2 で割ったときに余りが出ない整数のことです。
（2 で割り切れる整数ともいいます）
たとえば、2,4,6,8,10,・・・は偶数です。
偶数はすべて 2×〇 で表すことが出来ます。
（〇にはなにか整数が入ります）

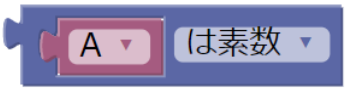
奇数とは、
2 で割ったときに 1余る整数のことです。
（2 で割り切れない整数ともいいます）
たとえば、1,3,5,7,9,・・・は奇数です。
奇数はすべて （2×〇)+1 で表すことが出来ます。
（〇にはなにか整数が入ります）

すべての整数は、かならず偶数か奇数のどちらかです。
なので
「A は偶数じゃない」と「A は奇数である」は
同じ意味となります。

変数A の値が「整数」かどうかを判定します。
整数であれば「はい」、整数でなければ「いいえ」と返します。

※整数とは

整数とは、小数でも分数でもない数のことです。
もっと詳しく言うと、
0 と 0 に1ずつ足したり引いたりすることで成り立つ数のことをいいます。
たとえば　6は　0+1+1+1+1+1+1　なので、整数です。
たとえば　2.1 は　0+1+1+0.1　なので、整数ではありません。
たとえば　-4は　0-1-1-1-1　なので、整数です。



変数A の値が「素数」かどうかを判定します。
素数であれば「はい」、素数でなければ「いいえ」と返します。

※素数とは

素数とは、1 もしくは その数自身でのみ割り切れる整数のことです。
（「約数が 1 とその数自身しかない整数」ともいいます。）
ちなみに、1 は 1 でしか割り切れないので素数ではありません。

たとえば、2,3,5,7,11,13,17,19,29,31,･･･は素数です。

たとえば　2 は　1 と2でしか割り切れないので、素数です。
たとえば　6 は　1 と 6 のほかに、2と 3 でも割り切れるので、素数ではありません。
たとえば　19 は　1 と 19 でしか割り切れないので、素数です。
たとえば　121 は　1 と 121 のほかに、11 でも割り切れるので、素数ではありません。
…と、素数である数はとてもたくさんあるので、
興味があればぜひ調べてみてください。

変数A の値が「3 で割り切れる」かどうかを判定します（3 の部分は他の数字に書き換えられます）。割り切れれば「はい」、割り切れなければ「いいえ」と返します。

変数A の値が「正の数」かどうかを判定します。
正の数であれば「はい」、正の数でなければ「いいえ」と返します。

変数A の値が「負の数」かどうかを判定します。
負の数であれば「はい」、負の数でなければ「いいえ」と返します。

※正の数・負の数と 0 について

正の数とは、
0より大きい数のことです。
これまでに紹介した数たちと違って、整数でないものも
0 より大きければ正の数となります。

0.00002 も、$\frac{1}{4}$ も 0 より大きいので正の数です。

負の数とは、
0より小さい数のことです。
これまでに紹介した数たちと違って、整数でないものも
0 より小さければ負の数となります。
正の数と区別するため、数字の左側に - という記号を付けます。

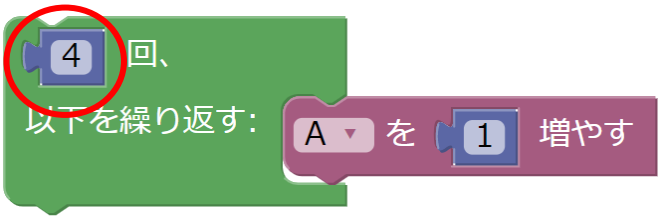
- 0.00002 も、 $-\frac{1}{4}$ も 0 より小さいので負の数です。

0 は正の数でも負の数でもありません。
なので、判定する条件の中に0も含みたい場合は
$\geqq 0$ や $\leqq 0$ を使うと良いでしょう。

8.　同じ処理をくりかえしたい＜くりかえしブロック＞

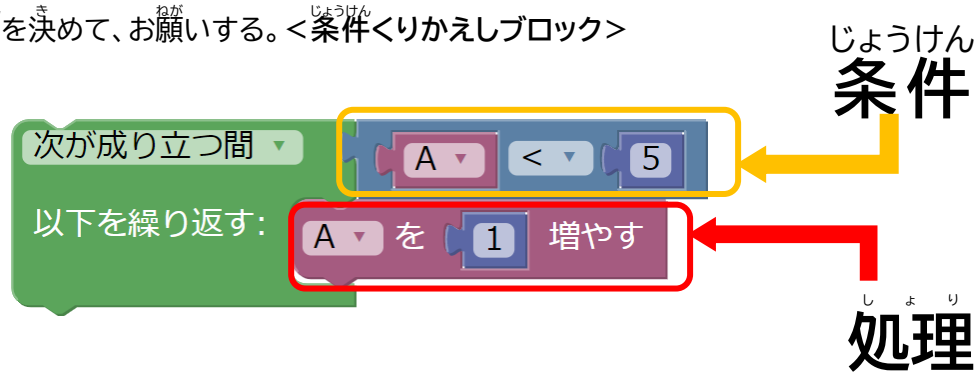
コンピュータは決まった処理を繰り返すことも得意です。
処理をお願いするときに、以下の決まりで同じ処理を繰り返してもらうことが出来ます。

① 回数を決めて、お願いする。＜**回数くりかえしブロック**＞

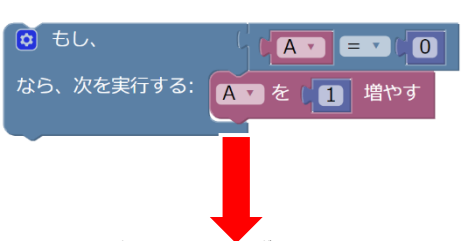


○で囲んだ部分で繰り返してほしい回数を指定できます。
この場合は**「変数A の値 を 1増やす」という処理を 4回繰り返します。**
変数A の値がもともと 0 だった時、このくりかえし処理を終えると変数A の値は4になっています。

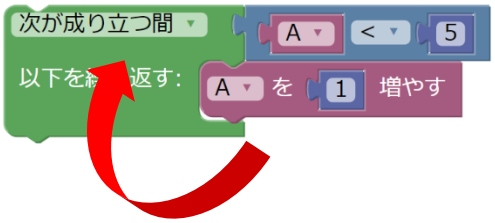
② 条件を決めて、お願いする。＜**条件くりかえしブロック**＞



＜**条件分岐ブロック**＞とおなじように、条件にあてはまっているときだけ処理をしてくれますが、＜**条件分岐ブロック**＞と違って、処理が終わったらまたブロックの前に戻ります。

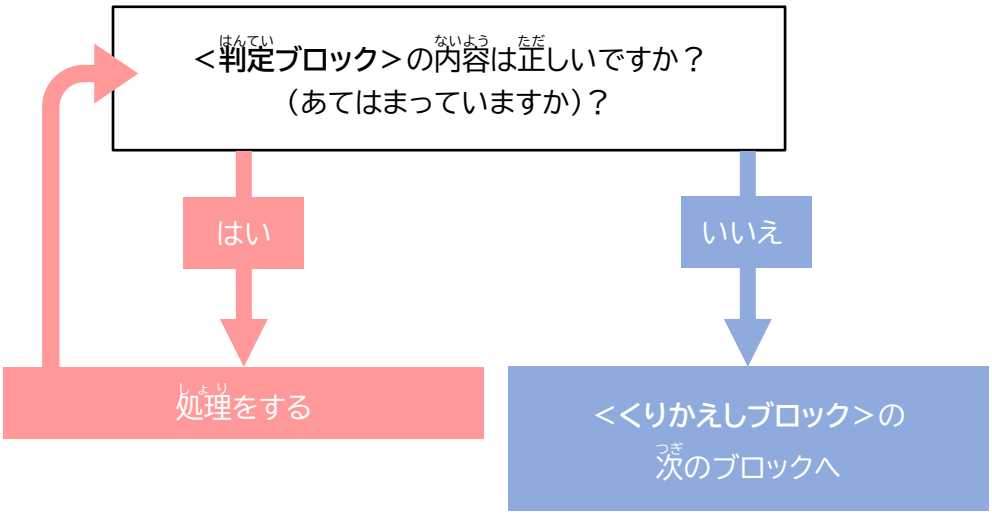


処理が終わったら**次のブロック**へ



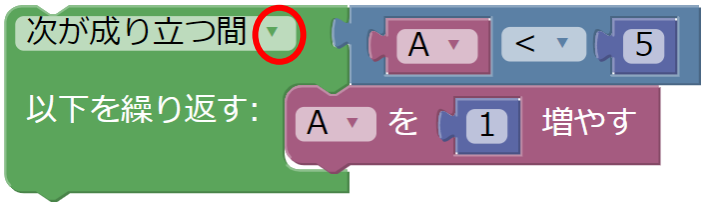
処理が終わったら**くりかえしブロック**の頭へ

この＜**条件くりかえしブロック**＞は次のような順番で判断と処理を繰り返します。

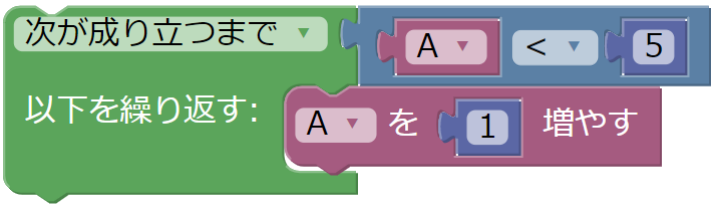


＜**条件くりかえしブロック**＞は処理が終わるともう一度、
「＜**判定ブロック**＞の内容は正しいか？（あてはまっているか）？」を考え直します。
そして、当てはまっていたら処理をする→また「＜**判定ブロック**＞の内容は正しいか？（あてはまっているか）？」を考える…　のくりかえしです。
これは＜**判定ブロック**＞の内容が正しくなくなるまで（あてはまらなくなるまで）、行います。

＜**条件くりかえしブロック**＞



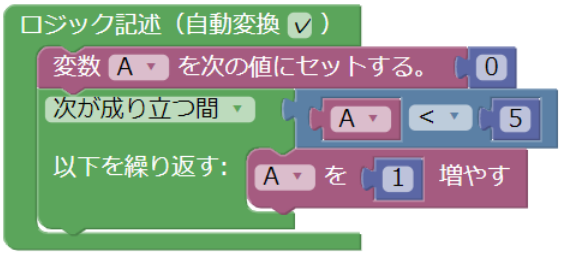
このブロックは「変数A の値 が 5 より小さい」という条件が成り立つ（あてはまる）間だけ、「変数A の値 を 1増やす」という処理をします。
このブロックは○のところをクリックして「次が成り立つ（あてはまる）まで」にも切り替えられます。



この2つの違いを詳しく説明すると…

「次が成り立つ間」は

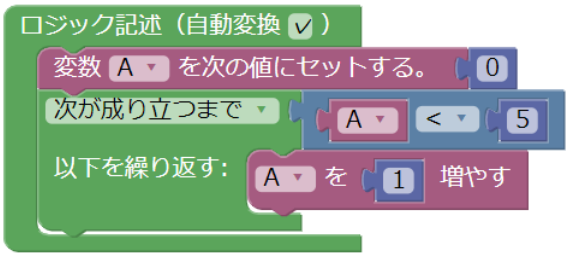
条件に当てはまっている間、
処理を行います。
あてはまらなくなったら、処理をせず次のブロックに行きます。



処理のながれ	変数A の値
① 変数A の値 を 0 にセットする。	0
②「変数A の値 が 5 より小さい」という条件に当てはまっているかを判定します。	0
変数A の値 は 5 より小さいので「あてはまっている！」と判断します。	
③「変数A の値 を 1増やす」 という処理をします。	1
④ ②に戻り、「変数A の値 が 5 より小さい」という条件に当てはまっているかを判定します。	1
変数A の値 は 5 より小さいので「あてはまっている！」と判断します。	
③「変数A の値 を 1増やす」 という処理をします。	2
④ ②に戻り、「変数A の値 が 5 より小さい」という条件に当てはまっているかを判定します。	2
変数A の値 は 5 より小さいので「あてはまっている！」と判断します。	
③「変数A の値 を 1増やす」 という処理をします。	3
④ ②に戻り、「変数A の値 が 5 より小さい」という条件に当てはまっているかを判定します。	3
変数A の値 は 5 より小さいので「あてはまっている！」と判断します。	
③「変数A の値 を 1増やす」 という処理をします。	4
④ ②に戻り、「変数A の値 が 5 より小さい」という条件に当てはまっているかを判定します。	4
変数A の値 は 5 より小さいので「あてはまっている！」と判断します。	
③「変数A の値 を 1増やす」 という処理をします。	5
④ ②に戻り、「変数A の値 が 5 より小さい」という条件に当てはまっているかを判定します。	5
変数A の値 は 5 より小さくないので「あてはまっていない！」と判断します。	
プログラム終了	

「次が成り立つまで」は

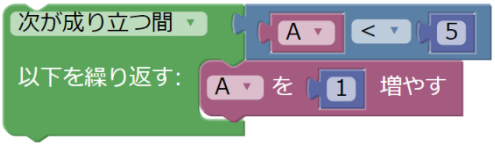
条件に当てはまっていない間、
処理を行います。
あてはまったら、処理をせず次のブロックに行きます。



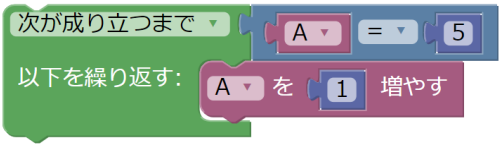
処理のながれ	変数A の値
① 変数A の値 を 0 にセットする。	0
②「変数A の値 が 5 より小さい」という条件に当てはまっていないかを判定します。	0
変数A の値 は 5 より小さいので「あてはまっている！」と判断します。	
プログラム終了	

「変数A の値 が 5 より小さい」という条件に当てはまっていないときに処理を行いますので、この場合は処理をせずにプログラムを終了してしまいます。

たとえば、



と



左側は「変数A の値 が 5 より小さい」という条件に当てはまっている間、「変数A の値 を 1 増やす」 という処理をします。

右側は「変数A の値 が 5 である」という条件に当てはまるまで、「変数A の値 を 1増やす」という処理をします。

どちらも最初に変数A の値 を5より小さい整数にセットしていれば、まったく同じ結果となります。

ですが、この＜**くりかえしブロック**＞より前の時点で
変数A の値 が 5 より大きくなったり、1.1 などの 1 をいくらたしても 5 にならないような値だったりする場合、右側の組み立て方では**大変なことになってしまいます！**

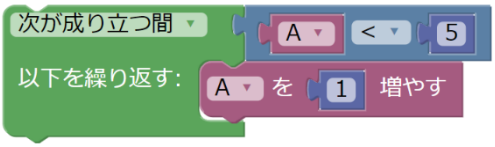
なんと、**無限ループになってしまうのです！！**

無限ループについて

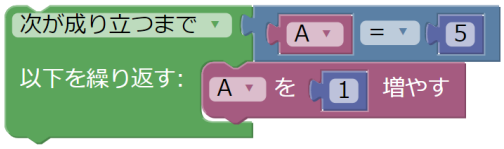
無限とは**限りがない**という意味で、ループとは**くりかえし**という意味です。
限りなく続くくりかえし…つまり無限ループとは、くりかえし処理から抜け出せずプログラムが終了出来ない状況のことです。
プログラムが終了できないと、コンピュータに大きな負担がかかったり、予想しない動きをしたりします。

そうならないように、プログラムを組み立てるときは細心の注意を払って、無限ループにならないよう工夫をしなければなりません。

先ほど、紹介したこの 2 つのブロック。



と



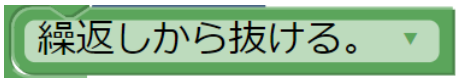
どちらも最初に変数A の値を5より小さい整数にセットしていれば、まったく同じ「変数A の値が 5」という結果となります。

でも変数A の値が 5 より大きくなったり、1.1 などの 1 をいくらたしても 5 にならないような値だったりする場合、左側のブロックは途中で止まりますが、右側のブロックは永遠に変数A の値に1をたし続けてしまいます！！

たとえば、変数A の値が 6 だったとき、いつまでたっても「変数A の値が 5 である」という条件に当てはまらないので、「変数A の値を 1増やす」 という処理をし続けます。
変数A の値は 6→7→8→9→10→･･･と増え続けるのです。
変数A の値が 1.1 などの小数だったときも、同じです。
変数A の値は 1.1→2.1→3.1→4.1→5.1(5 じゃないととまらない)→6.1→･･･と増え続けるのです。

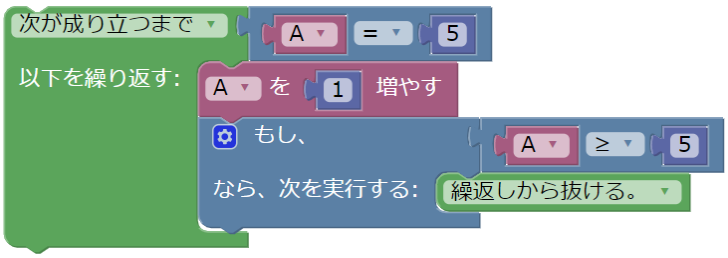
もともと、**変数A の値は5より小さい整数になるはずで、「変数A の値が 5」という結果がほしい**のであれば、左側の組み方でプログラミングした方がよいのです。
でも、右側のような条件のほうがプログラムとして組みやすい場合もあります。
そんなときは…

＜**繰返しから抜けるブロック**＞を使います。



○のところをクリックして「繰返しの次の反復処理を続行する」にも切り替えられます。

これは＜**くりかえしブロック**＞の次のブロックへいきます。
つかうときは

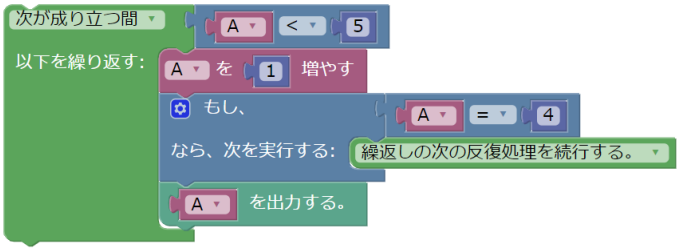


このように、＜**くりかえしブロック**＞のなかに＜**条件分岐ブロック**＞をいれて、その処理部分にいれるとよいです。
こうすることで、変数A の値が 5 より大きくなったら＜**くりかえしブロック**＞の次のブロックへいくので、無限ループにはなりません。

無限ループを防ぐものではありませんが、

＜**繰返しの次の反復処理を続行するブロック**＞というものもあります。

これは、このブロックのあとにある処理を行わず、また条件の判定にもどるブロックです。
特定の場合だけ処理をしたくないときに使います。
たとえば…



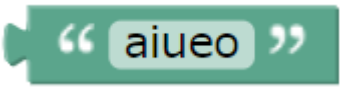
1
2
3
5

こんなふうに出力されます

「変数A の値が 5 より小さい」という条件に当てはまっている間、「変数A の値を 1増やす」と「変数A の値を出力する」いう2つの処理を行いますが、「変数A の値が 4」のときは「変数A の値を出力する」という処理をせず、条件の判定にもどります。

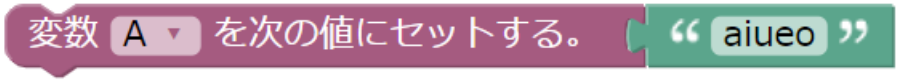
9. 文字列を使いたいとき

＜**文字列ブロック**＞を使えば、アルファベット、ひらがな、カタカナ、漢字も使うことが出来ます。

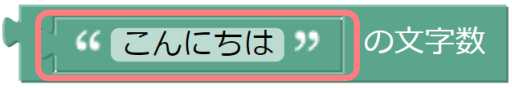


＜文字列ブロック＞
好きな文字列を入力することが出来ます。

＜**数をつくるブロック**＞などと同じように、＜**変数ブロック**＞などとつなげたりして使います。

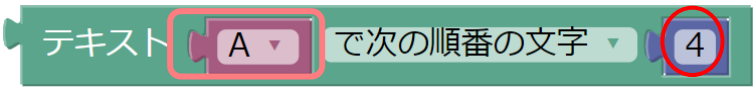


このようにつなげます。すると、変数A の値は「aiueo」となります。
＜**変数ブロック**＞にセットすることで、問題文や解答文から数字や文字列と同じように{A}で呼び出すことが出来ます。もちろん、変数名はなんでもよいです。

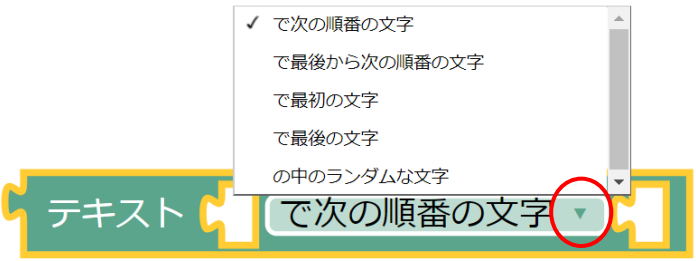


＜文字数ブロック＞
文字列がなん文字かを数えてくれます。
この場合は「こんにちは」と 5文字なので、5 という値を返してくれます。
□ で囲んだところは変数ブロックでもはめることが出来ます。

＜文字列の中の文字ブロック＞



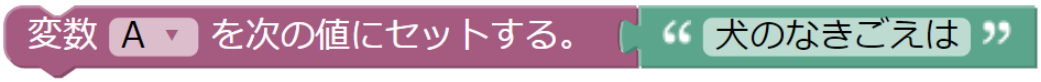
□ で囲んだところにはめたブロックの文字列の中の、特定の文字だけ取り出すことが出来ます。このブロックは頭から 4番目の文字を取り出します。



文字列が「あいうえお」で、順番が 4 のとき	
①	え
②	い
③	あ
④	お
⑤	あ、い、う、え、お のどれか

このブロックは○をクリックすると
「①(文字列)の次の順番の文字」、
「②(文字列)の最後から次の順番の文字」、
「③(文字列)の最初の文字」、「④(文字列)の最後の文字」、
「⑤(文字列)の中のランダムな文字」…にも変更できます。

<**文字列ブロック**>を次のような内容のものに付け替えると

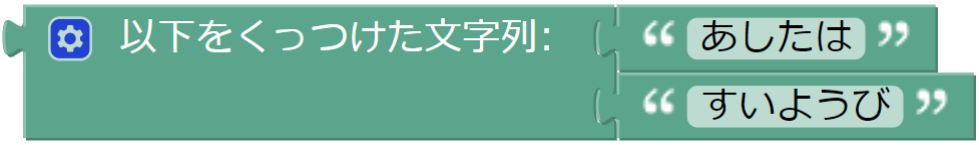


変数A の値は書き換えられて、「犬のなきごえは」となります。

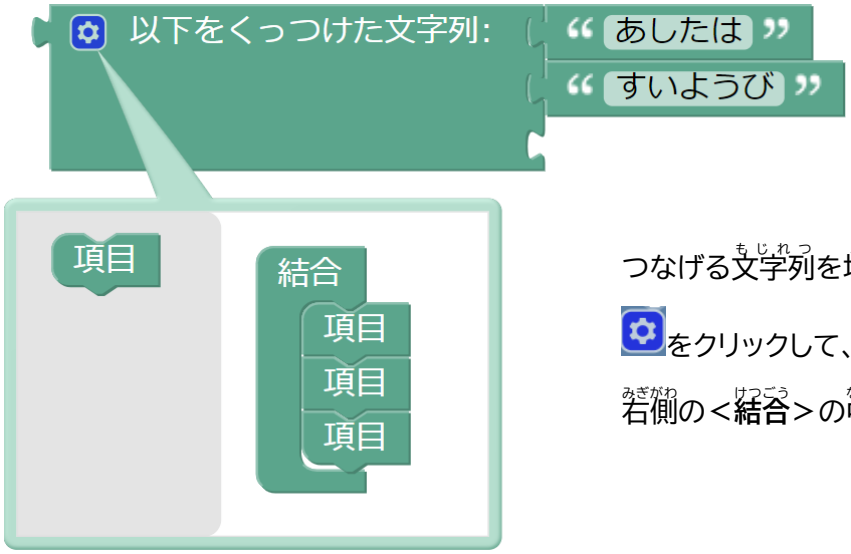
変数A の値は書き換えずに、文字列を追加できる<**文字列追加ブロック**>もあります。

このブロックを使うと、変数A の値は「犬のなきごえは」に**「わんわん！」が追加さ**れて、**「犬のなきごえはわんわん！」**となります。

追加ではなく、2 つ以上の文字列をつなげて変数の値にセットしたいときは
<**文字列結合ブロック**>を使います。



この場合は「あしたはすいようび」という文字列になります。



つなげる文字列を増やしたいときは、

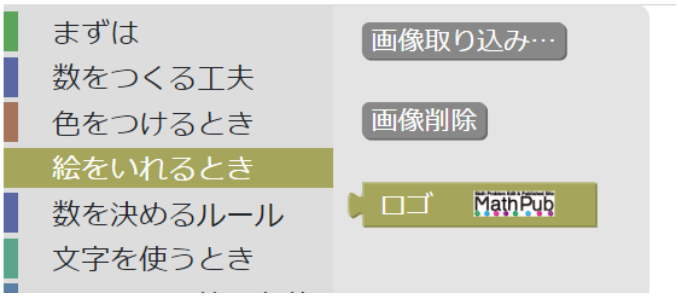
をクリックして、<**項目**>を
右側の<**結合**>の中につなげます。

10.　絵を使いたいとき

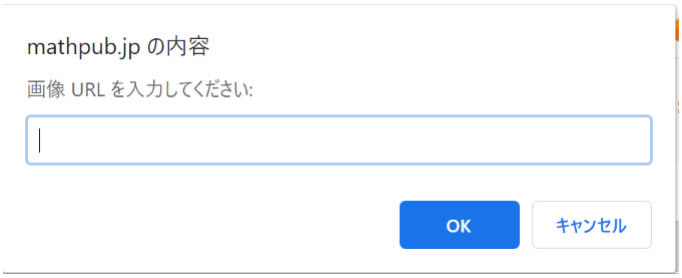
図形問題などで絵(画像)を使いたいときは＜**画像ブロック**＞を使います。

このブロックはWebアドレスがある画像しか使えません。
ＵＲＬアドレスを指定して使いますので、インターネットなどで見つけた著作権フリー画像などは、配布元のWebサイトが「直リンクでの使用」を許可しているかどうかをご確認ください。

MathPub Centerが用意している図形Webサイト「教材に使う図形一覧」
（ https://mathpub.jp/img/index.html ）というページにある画像については、教材作成に限り自由に利用することが出来ますので、ぜひお使いください。

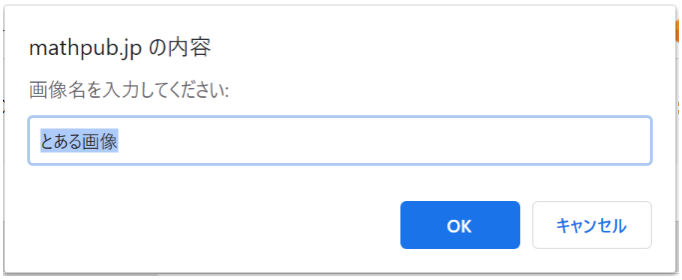


まずは、「画像取り込み…」をクリックします。

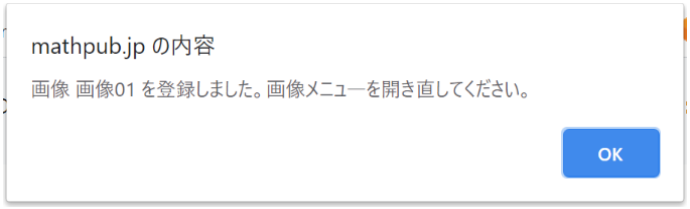


すると画像のＵＲＬを入力するダイアログ（画面の上部に表示されるメッセージ）が出てきますので、そこに、使いたい画像のＵＲＬを入力します。

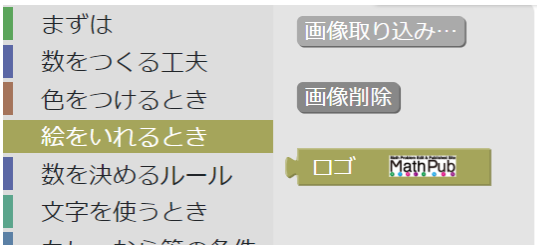
ここでは、
「教材に使う図形一覧」の
円の面積を求める画像　円(1)を
使うことにしました。



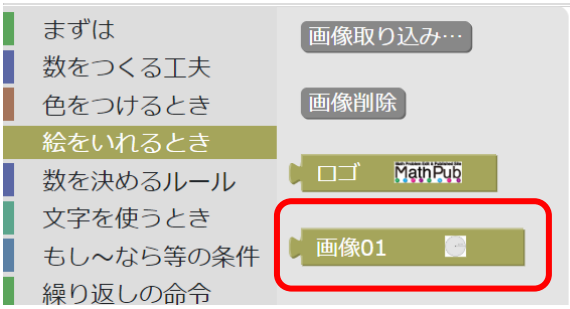
ＵＲＬを入力すると、
次は画像の名前を入力します。
これがそのままブロックの名前となりますので、わかりやすい名前を付けましょう。



と、いいながら
「画像01」という名前を付けました。

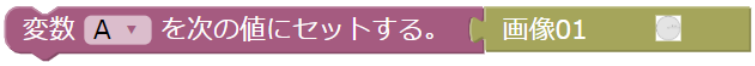


名前を付けると、ダイアログがきえます。
このままでは何も増えていませんが、
他のメニュー（例えば「数を決めるルール」など）を開いてから…



もういちど「絵をいれるとき」メニューを開くと、先ほど作成した
＜画像01ブロック＞が
増えています！

使うときは、

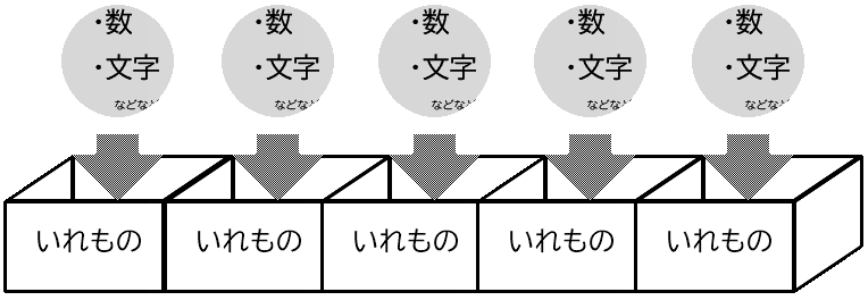


このように**＜変数ブロック＞**にセットすることで、問題文や解答文から数字や文字列と同じように**{A}**で呼び出すことが出来ます。

もちろん、変数名はなんでもよいです。

11.　リストについて

リストとは、変数のようないれものがいくつかセットになっているものです。



この連なっているすべてのいれものをまとめて **リスト** とよびます。
このイメージ図では 5個ですが、個数は何個あってもよいです。

何種類かあるデータや値を、1 つのかたまりとして使いたいときに便利です。

たとえば、

「今日のラッキーフルーツは{くだもの}です。」
※**{くだもの}**には **りんご、ぶどう、みかん、もも、いちご** のなかから、
ランダムにひとつ選んで表示するようなプログラムを作ります。

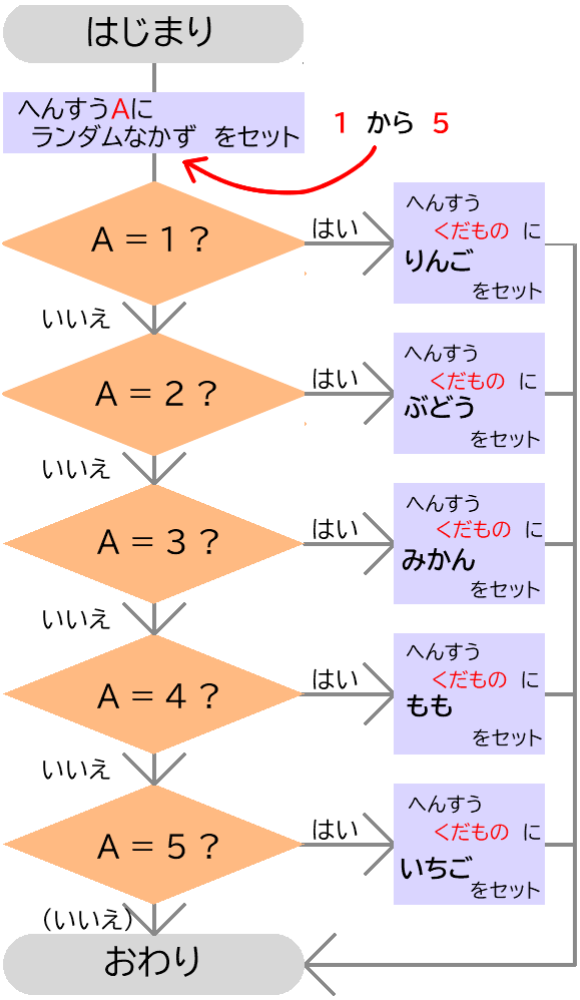
作るにはいくつか方法がありますが、今まで紹介してきたブロックを考えると

ランダムにひとつ選ぶ…というのは、
まず、**＜ランダムな数をつくるブロック(乱数ブロック)＞**を使って数字を1つ作り、
1 がでたら**りんご**、2 がでたら**ぶどう**、3 がでたら**みかん**、4がでたら**もも**、5がでたら**いちご**
を**変数くだもの**にセットするようにプログラミングすると実現できます。

でてきた数字によってセットする値を変える…条件によって処理を変える…
これは**＜条件分岐ブロック＞**を使うとよさそうです！

さっそくプログラミングしてみましょう。

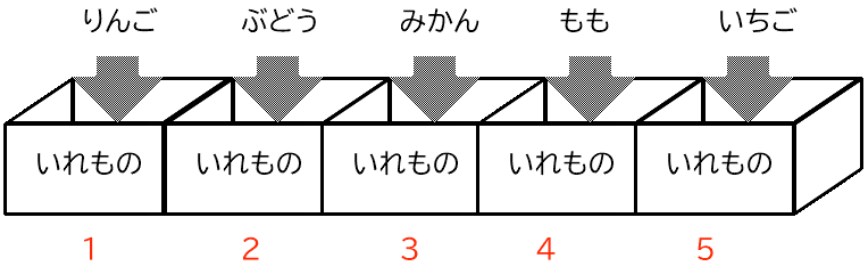
ながれを図に描いてみました(ながれを図で表したものをフローチャートとよびます)。



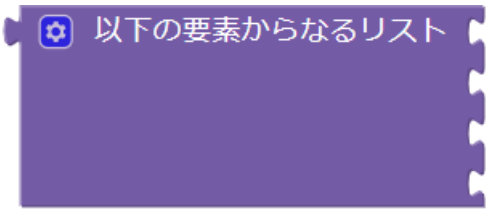
ブロックの組み方はこのような感じです。

なのですが。 リストをつかうと、このプログラムをもっとコンパクトにすることが出来ます。

まずは＜**リストブロック**＞を紹介しましょう。
たとえば、この図のような リスト(いれもののかたまり)を作ります。



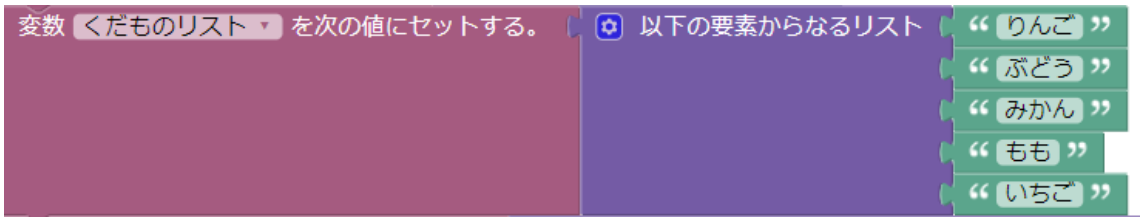
「リスト」メニューにある、＜**以下の要素からなるリストブロック**＞を使います。



 をクリックすると、＜**文字列結合ブロック**＞のようにつなげられる項目数を増やしたり、減らしたり出来ます。

このブロックに、くだもの(りんご、ぶどう、みかん、もも、いちご)をセットします。

そして、この＜**リストブロック**＞を＜**変数ブロック**＞につなげてあげましょう。

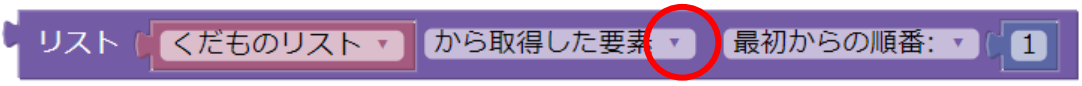


今回、このリストの名前は **くだものリスト** としました。
何でもよいのですが、わかりやすい名前にしましょう。

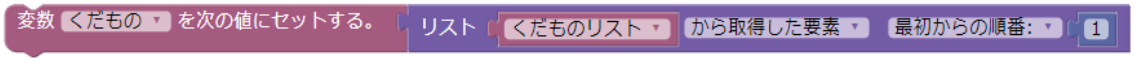
このリストの良いところは、いれものひとつひとつに番号がふってあるので、
リスト〇〇の何番目！といえば、そのまま中身を取り出すことが出来るところです。

リストの中身、ひとつひとつのことを要素といいます。
先ほどの例だと **りんご**や**ぶどう**などは「リスト **くだものリスト** の **要素**」です。

＜リスト・要素取得ブロック＞



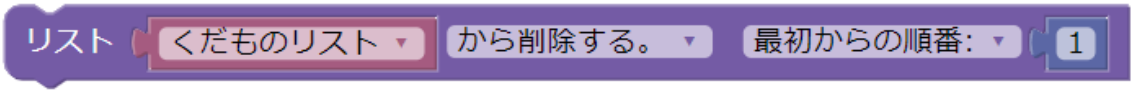
リストにセットされている要素をひとつ、取り出すことが出来るブロックです。
イメージとしては「リスト〇〇の△番目の要素を教えて！」とお願いする感じです。
この例だと、リスト **くだものリスト**の最初から数えて 1番目の要素を教えてくれます。
1番目ってなんでしたっけ…？　　そう、**りんご**です。



このようにブロックを組むことで、変数**くだもの** の値が りんご となります。

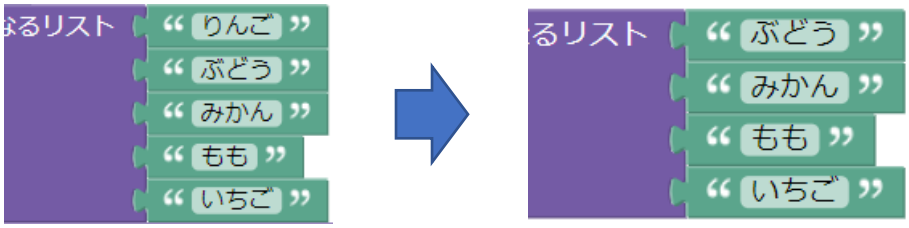
〇をクリックすると、以下のブロックに変更できます。

＜リスト・要素削除ブロック＞



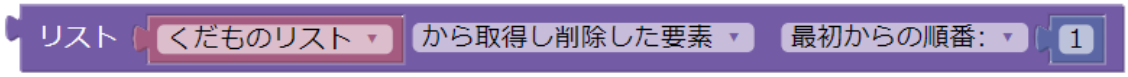
リストにセットされている要素をひとつ、削除するブロックです。
この例だと、リスト **くだものリスト**の最初から数えて 1番目の要素を削除します。

リスト **くだものリスト**の要素数も減りますので、他のブロックで「リストの△番目～」と
指定している場合は気を付けてください。

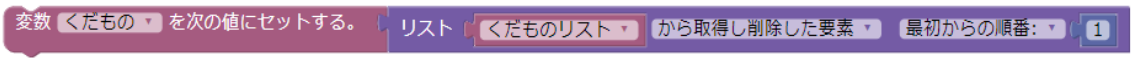


りんごが消えた！

＜**リスト・要素取得して削除ブロック**＞

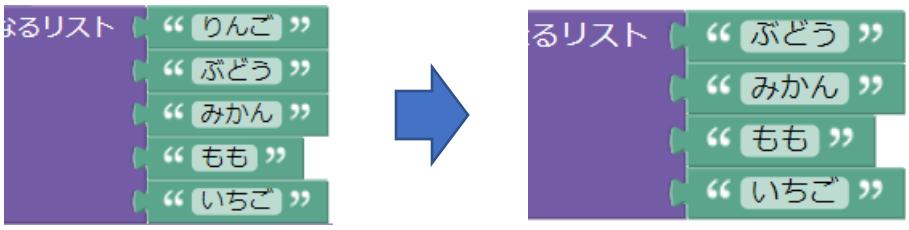


リストにセットされている要素をひとつ取り出して、さらにその要素を削除します。
この例だと、リスト **くだものリスト**の最初から数えて 1番目の要素を教えてくれて、
さらに 1番目の要素を削除します。



このようにブロックを組むことで、変数**くだもの** の値が りんご となります。

リスト **くだものリスト**の要素数も減りますので、他のブロックで「リストの△番目～」と
指定している場合は気を付けてください。

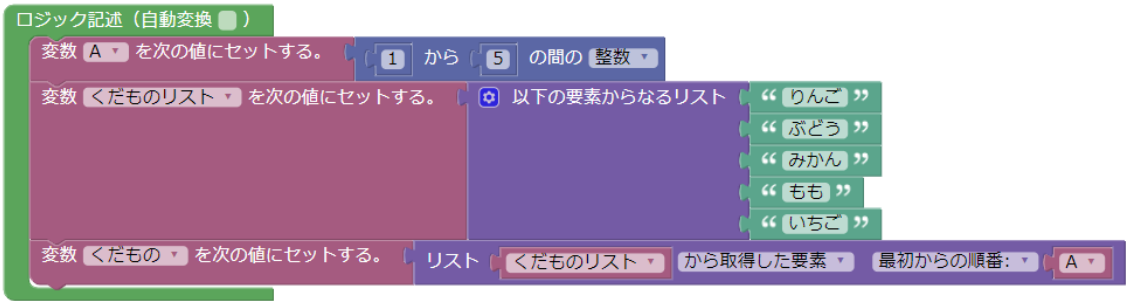


りんごが消える。

これらのブロックを使うと＜**条件分岐ブロック**＞を使わずに

「今日のラッキーフルーツは{くだもの}です。」
※**{くだもの}**には **りんご、ぶどう、みかん、もも、いちご** のなかから、
ランダムにひとつ選んで表示するようなプログラム
を作ることが出来ます。

ブロックの組み方はこのような感じです。

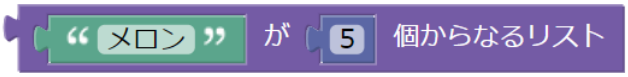


どうでしょうか…こちらのほうがすっきりしていませんか？

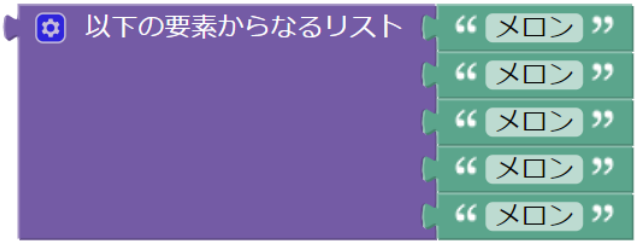
＜空のリスト作成ブロック＞

要素数も決まっていない、空っぽのリストを作成します。
先に紹介した**＜リストの要素挿入ブロック＞**と合わせて使います。

＜同じ要素のリスト作成ブロック＞

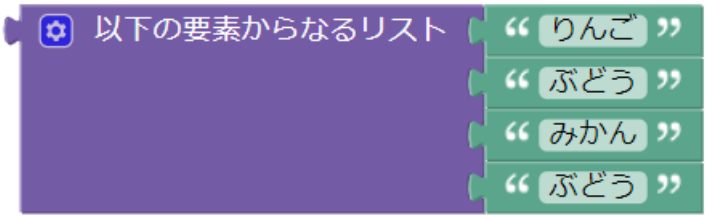


すべて同じ要素でうめたリストを作成します。
この場合は「メロン」という要素が 5個入っているリストを作成します。

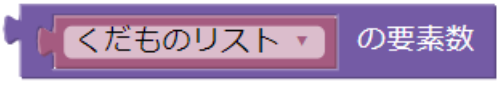


こんな感じのリストを作成してくれます。

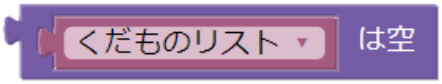
リストを操作するためのブロックは他にもあります。



この先は、この画像のようなリスト **くだものリスト** を使って、説明していきます。

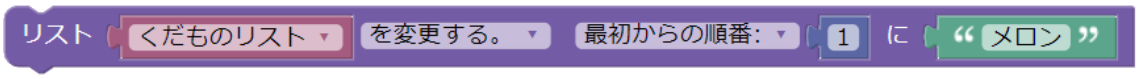


＜リストの要素数ブロック＞
そのリストにいくつ要素があるのかを教えてくれます。この場合は 4 という数になります。

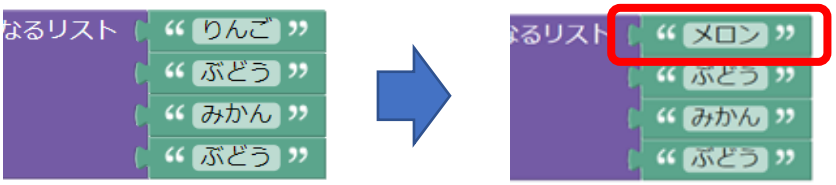


＜リストが空か判定ブロック＞
そのリストに要素がひとつもない状態かどうかを判定します。

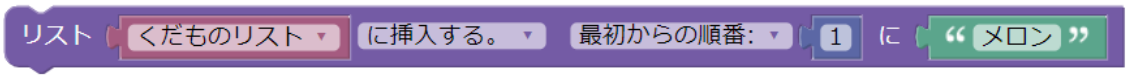
＜リストの要素変更ブロック＞



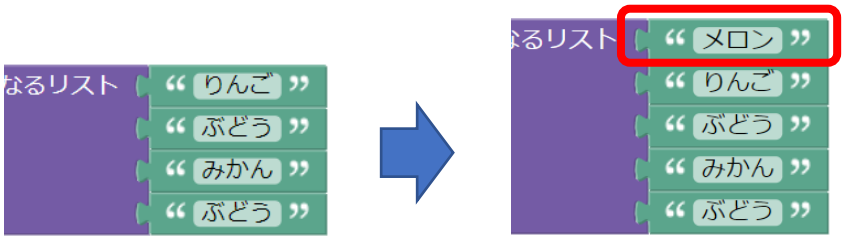
リストの決められた位置の要素を変更します。この場合は、**りんご**が**メロン**に変わります。



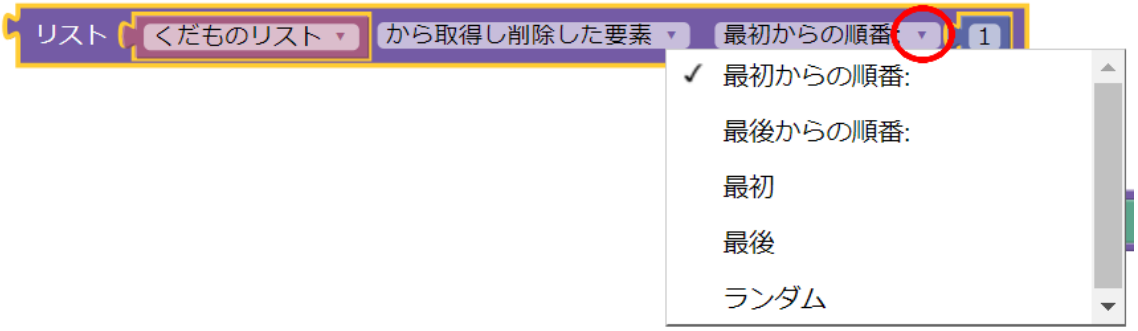
＜リストの要素挿入ブロック＞



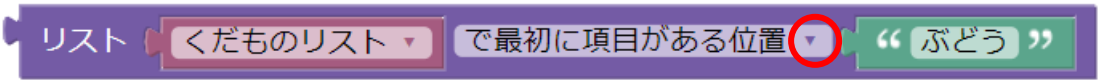
リストの決められた位置の要素を挿入(追加)します。この場合は、**1つめの要素**として**メロン**が追加となり、**りんご**や**ぶどう**は1つずつうしろにずれます。



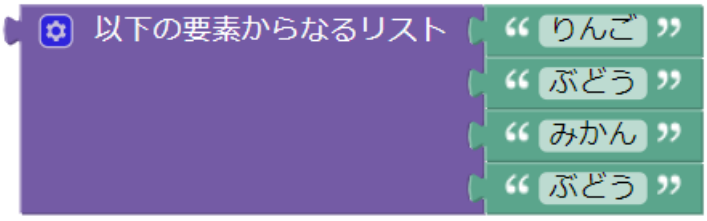
「最初からの順番」とあるところは、○のところをクリックすると、
「最後からの順番」「最初」「最後」「ランダム」にも変更できます。



＜リストの要素位置ブロック＞

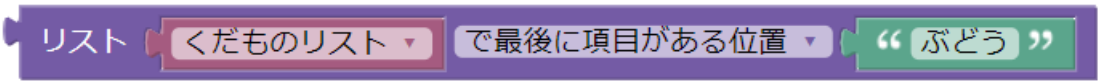


この**くだものリスト** のなかで、「ぶどう」という要素は、リストを先頭からみていくと最初にでてくるのはいくつめですか？ について教えてくれます。



この場合は、1つめの「りんご」から見ていくので、最初に「ぶどう」が出てくるのは2つめになります。なので、「2」という数が答えとなります。

また、○をクリックすると「最後に項目がある位置」に変更できます。



この**くだものリスト** のなかで、「ぶどう」という要素は、リストを先頭からみていくと最後にでてくるのはいくつめですか？ について教えてくれます。

「ぶどう」は先頭から 2 つめと 4 つめにありますが、最後なのは 4 つめですので「4」という数が答えとなります。

さて、以上でブロックのご紹介は終わりです。
ぜひ、いろんなブロックを使ってみてください。

次のページからは、プログラミングするうえでの注意点や心がまえについてお話しします。

プログラミング的思考を養うために

プログラミングをするうえで、必要なことはなんでしょうか…

それは、コンピュータとコミュニケーションをしっかりとれることです。

コンピュータが出来ることを理解して、やりたいことを、コンピュータが出来ることに分解して組み立てる。
分解の仕方、組み立てた結果、見通しがよく、無駄が少ないのがよいプログラムといえると思います。

MathPubの場合は、1つの教材の中に「問題の意図」や「解答の意図」、「プログラムの意図」という欄があり、その内容がきちんと反映されているか…というところに注意してください。

もちろん、自分が新しく教材を作るときは、意図(つまり、こまかい設計ポイント)が人間にも、コンピュータにもわかりやすく書かれていることが大事です。

また、まったく同じ結果をだすプログラムだとしても、
・変数の割り当て方
・条件分岐の使い方
・くりかえし処理の使い方
・入れ子構造の順番
・処理の順番
…などが違ったりして、組み立て方は様々です。

ですが、プログラミングの世界では**シンプルで、他の人がわかりやすく、コンピュータへの指示(この場合はブロックの量)が少ない方がよいプログラムだ**といわれています。

「なるべく少ない手数で」、やりたいことが出来るようなプログラムがカッコイイ！のです。

みなさんも慣れてきたら、ぜひ**シンプルで、他の人がわかりやすく、コンピュータへの指示(この場合はブロックの量)が少ないプログラム**になるよう心がけてみてください。