半日で覚えるMathematica

理数系教材に必要な基本機能

神戸大学 人間発達環境学研究科 長坂耕作

半日で覚えるMathematica

理数系教材に必要な基本機能

神戸大学 人間発達環境学研究科 長坂耕作

ノートブックやドキュメントセンターの使い方は含まれていません。セルやセルブラケットなども含め，これらの名称や操作方法については，「ノートブックの基本」を *Mathematica* 上で参考にしてください。講習会や研修会などでの場合は，通常，開始直後に説明が行われます。

なお，このドキュメントも *Mathematica* のノートブックと呼ばれるファイル形式で作成し，最終的にPDF形式に出力したものを印刷配布しています。

【目次】

LECTURE01: 組込み関数の基本と電卓として使おう

Mathematica をプログラミング言語と考えたときの文法と，関数電卓と同じような計算をさせるにはどのようにすれば良いのか，という基本的なことを学びます。この章の到達目標は，次のような計算を *Mathematica* にさせることになります。

- サイコロを100回振ったときの目の合計と平均を求めましょう。
- 円周率を500桁求めましょう。
- 円の面積と，それを三角形分割して計算したときの値を比較しましょう。

四則演算と組込み関数

Mathematica で「2足す3」を計算させるには，次のように入力して評価します。

```
Plus[2, 3]
```

```
5
```

Mathematica に入力した式を実際に評価して計算させるには，当該セルのどこでも良いので，「SHIFT+ENTER」を入力します。つまり，シフトキーを押しながらエンターキーを押します。

このとき，Plusを関数，2と3を引数と呼びます。関数と引数の間と最後には，大括弧「[」と「]」を入力します。引数を変更すれば，違う計算を行うことも出来ます。例えば，次の例は，「5足す12」を行っています。

```
Plus[5, 12]
```

```
17
```

では，「1と2と3を足す」にはどうしたら良いでしょうか。この場合，原理的には次のように「「1足す2」足す3」を行います。

```
Plus[Plus[1, 2], 3]
```
```
6
```

このように， *Mathematica* への命令は複数を組み合わせる（または，ネストさせる）ことが出来ます。しかし，より便利な使い方があります。それは，次のように引数を単純に3つにすることです。

```
Plus[1, 2, 3]
```
```
6
```

このように，演算が順序に依らずに計算結果が変化しない（結合律が成り立つ）ときは，関数をネストしなくても使えるようになっています。*Mathematica* に入力した式を実際に評価して計算させるには，当該セルのどこでも良いので，「SHIFT+ENTER」を入力します。つまり，シフトキーを押しながらエンターキーを押します。*ematica* ではこのような仕組みを利用できる関数の属性のことを「Flat」と呼びます。関数Plusがこの属性を持つかは，「Attributes[Plus]」で確認できます。

演習

加減乗除は，それぞれ Plus, Subtract, Times, Divide に対応します。それぞれを使って以下の演算を Plus[1, 2] のような表現で計算してみてください。また，もし Flat な属性を用いて計算可能ならば，それも使って下さい。

(1) 1 + 2 + 3 + 4 + 5

(2) 5 − 2

(3) 5 × 4 × 3 × 2

(4) 6 ÷ 3

リストと四則演算

「1足す2」を行う場合，次のように関数を使うことを習いました。

```
Plus[1, 2]
```
```
3
```

このとき，関数に渡される引数の集合{1, 2}を考えます。

```
{1, 2}
```

Mathematica では，このように中括弧「{」と「}」で括ったものをリストと呼びます。リストの要素はカンマ「,」で区切ります。リストは，集合やベクトル，行列，テンソルなど様々な数学的な対象を表現するのに使われます。次の例は，集合{1, 2, 3}を表しています。

```
{1, 2, 3}
```

このような引数の集合に対して，Plusなどの関数を適用するにはどのようにすれば良いでしょうか。単純に，「{」を「[」に，「}」を「]」に変更し，前にPlusを付けても計算は可能です。

```
{1, 2, 3}
```

```
Plus[1, 2, 3]
```

```
6
```

Mathematica ではもっと簡単な方法が用意されています。上で行った操作（中括弧を大括弧に書き直し，関数を前に付ける）と同じことをアットマーク二個「@@」で行うことが出来ます。これを関数をリストに適用（Apply）すると言います。

```
Plus @@ {1, 2, 3}
```

```
6
```

演習

加減乗除は，それぞれ Plus, Subtract, Times, Divide に対応しました。下記の演算を引数の集合をリストを使って表し，それに関数をApplyする方法で計算をしてみてください。

(1) 1 + 2 + 3 + 4 + 5

(2) 5 – 2

(3) 5×4×3×2

(4) 6÷3

より数学らしく四則演算を行う

ここまでは，*Mathematica* の関数とリストの使い方を説明するために，四則演算を少し難しい方法で行っていましたが，通常通りに入力すれば，*Mathematica* はそのまま計算を行ってくれます。

```
1 + 2 + 3 + 4 + 5
```

```
15
```

Mathematica では，加算にプラス「+」を，減算にマイナス「−」を，乗算にアスタリスク「*」を，除算にスラッシュ「/」を，冪乗にキャレット「^」を使います。小括弧「(」と「)」を使って，演算順序を指定することも可能です。

```
1 * 2 * 3 - 4 + 4 * (5 - 3) + 6 / 2
```

```
13
```

半角のスペースは乗算を意味しますので，注意してください。また，小括弧，中括弧，大括弧にはそれぞれ役割が決まっていますので，演算順序を指定する場合は，すべて小括弧を使わなければいけません。

冪乗は関数Powerを使っても計算できます。

```
2^3
```

```
8
```

```
Power[2, 3]
```

```
8
```

```
Power @@ {2, 3}
```

```
8
```

ここまで気づかれたと思いますが，*Mathematica* の組込み関数は全て大文字から始まります。小文字と大文字の違いを認識しますので，注意して入力してください。

また，ここまで整数の範囲で行える演算のみを扱ってきましたが，*Mathematica* で扱える数の種類には以下のようなものがあります。

- 整数（とその複素数への拡大）

```
-2, 1, 0, 1, 2, -2 + 3 i, 1 - 5 i, i
```

- 有理数（とその複素数への拡大）

$$\frac{2}{3},\ \frac{4}{9},\ \frac{12}{7},\ \frac{2+3i}{5},\ \frac{i}{4}$$

- 実数（とその複素数への拡大）

$$2\sqrt{2},\ 3\sqrt{5},\ e,\ \pi,\ \pi^{i}$$

- 浮動小数点数（とその複素数への拡大）

```
1.234, -0.99999, 0.543 + 1.3 i
```

虚数単位「i」は「[ESC]iii[ESC]」で，自然対数の底「e」は「[ESC]ee[ESC]」で，円周率「π」は「[ESC]pi[ESC]」で入力できます。それぞれ，「I」，「E」，「Pi」と入力しても同じ意味になります。

これらの数は，*Mathematica* では次の4種類に分類されます。

- Integer: 任意の桁数の整数
- Rational: 約分した分数，有理数
- Real: 指定された桁精度で近似された実数，浮動小数点数
- Complex: 数+数iの形の複素数

数の種類は，関数Headを使って確認することが出来ます。

```
Head[29]
```

```
Integer
```

```
Head[3.1415]
```

```
Real
```

演習

実際に複数の種類の数同士の演算を行って，結果がどのような種類の数になるかを確認してください。また，確認された事実から言えることをまとめて下さい。

$\sqrt{3}$ は，「CTRL+2」の後で「3」を入力することで，$\frac{1}{2}$ は，「1」の後で「CTRL+/」を押して「2」を入力することで，数学の慣用的な表現方法で入力することが出来ます。

厳密な数と近似的な数の行き来

厳密な数である「整数」，「有理数」や「実数」から「浮動小数点数」への変換には、次のように関数*N*を用います。*N*は Numeric の頭文字と覚えておくと良いでしょう。

```
N[1/2]
```

```
0.5
```

```
N[π]
```

```
3.14159
```

```
N[π, 20]
```

```
3.1415926535897932385
```

関数について詳しく知りたい場合は，関数を選択してファンクション

キー「F1」を押してください。ドキュメントセンターでの検索結果が自動的に表示されます。

この逆変換である浮動小数点数の有理数への変換は，Rationalizeを使用します。

```
Rationalize[0.5]
```

$$\frac{1}{2}$$

```
Rationalize[3.14159]
```

$$\frac{314\,159}{100\,000}$$

また，複数の厳密な数を同時に近似数に変換，もしくは，複数の近似数を同時に有理数に変換したいときは，次のようにアットマークを一つ「@」で関数を指定します。このように， *Mathematica* では複数のものを表すときにもリストを使います。

```
N@{E, Pi, GoldenRatio}
```

```
{2.71828, 3.14159, 1.61803}
```

```
Rationalize@{0.5, 0.333, 1.000}
```

$$\left\{\frac{1}{2}, \frac{333}{1000}, 1\right\}$$

「@」は，以前に出てきた「@@」とは異なります。キーボードから入力するときに，引数の最後に閉じ大括弧「]」を入力するのが面倒ときに利用できる「前置形」と呼ばれる入力方法になります。

もちろん，次のように大括弧を使って命令を行うことも可能です。

```
N[{E, Pi, GoldenRatio}]
```

```
{2.71828, 3.14159, 1.61803}
```

```
Rationalize[{0.5, 0.333, 1.000}]
```

$$\left\{\frac{1}{2}, \frac{333}{1000}, 1\right\}$$

Mathematica の関数で属性Listableを持っている場合，引数にリストが指定された場合，リストの各要素に同じ関数を展開して実行してくれます。関数Rationalizeがこの属性を持つかは，「Attributes[Rationalize]」で確認できます。

■ 数学関数を使おう

Mathematica には様々な数学関数が用意されています。その全ての紹介はドキュメントセンター（ヘルプ）に任せるとして，ここでは良く使われると思われる関数だけを紹介します。

対数関数と指数関数（詳細は関数名を選択してF1を押してください）

```
{Log[3], Log[3.], Exp[2], Exp[2.], Sqrt[3]}
```

$$\{\text{Log}[3], 1.09861, e^2, 7.38906, \sqrt{3}\}$$

三角関数（詳細は関数名を選択してF1を押してください）

```
{Sin[π/2], Cos[π/3], Tan[π/4], Sin[3]}
```

$$\left\{1, \frac{1}{2}, 1, \text{Sin}[3]\right\}$$

演習の解答になりますが，*Mathematica* は命令を厳密な数で与えられた場合，計算結果を厳密な数で返します。一方，命令の中に近似数が含まれている場合は，計算結果を近似数で返します。従って，上記のLogの例で，「3」と厳密な値を指定した場合と，「3.」と近似数で与えた場合では結果が異なっています。

階乗関数（詳細は関数名を選択してF1を押してください）

```
{2!, 3!, 4!, 5!}
```

```
{2, 6, 24, 120}
```

```
{Binomial[4, 2], Binomial[5, 2], Binomial[10, 3]}
```

```
{6, 10, 120}
```

数値関数（詳細は関数名を選択してF1を押してください）

```
Abs@{-1, -3, 4, 3.14, π, -e}
```

```
{1, 3, 4, 3.14, π, e}
```

```
Sign@{-5, π, -e, 10, 1/2}
```

```
{-1, 1, -1, 1, 1}
```

```
Round@{3.14, -3.14, 3/5, π, e}
```

```
{3, -3, 1, 3, 3}
```

■ 式の簡単化

複雑な三角関数からなる数式も変換を繰り返すうちに，とても簡単な式になる場合があります。そのような数式の簡単化を *Mathematica* は実行することが出来ます。使用する関数は，SimplifyとFullSimplifyです。二つの違いはドキュメントセンターで確認してください。

```
Simplify[Sin[2]/Cos[2]]
```

```
Tan[2]
```

```
FullSimplify[i Sin[1] + Cos[1]]
```

$$e^{i}$$

これら簡単化は三角関数のみならず，どのような数式にも適用可能です。

```
FullSimplify[Sqrt[1/2 (5 - Sqrt[5])] + Sqrt[1/2 (5 + Sqrt[5])]]
```

$$\sqrt{5 + 2\sqrt{5}}$$

より高度な使い方は次章（LECTURE02）で扱います。

■ リストの作り方（たくさんの数の作り方）

1から10までの和を計算する場合は，次のように入力しました。

```
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
```

```
55
```

しかし，これが100までであったり1000までであったりすると手で入力するのは大変です。そこで， *Mathematica* には便利な関数Rangeというのが用意されています。

```
Range[10]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

このように，Rangeを使うと簡単に特定の範囲の数を作り出すことが出来ます。従って，1から1000までの和を計算するには，次のように入力します。

```
Plus @@ Range[1000]
```

```
500 500
```

偶数だけの和を計算したい場合は，次のように引数を増やしてRangeを使います。詳しくは，ドキュメントセンターで確認してください。

```
Plus @@ Range[2, 1000, 2]
```

250 500

より高度な数の作り方としては，Rangeの機能を拡張したTableという関数を使う方法があります。まずは，Rangeと同じように数を作ってみます。

```
Table[i, {i, 1, 10}]
```

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

```
Table[i, {i, 2, 10, 2}]
```

{2, 4, 6, 8, 10}

これらの結果はRangeと同じなので，わざわざ別の関数を使うメリットはないように見えます。しかし，Tableでは次のように第一引数である「*i*」の部分に任意の数式を記述することが出来ます。これにより，指定した規則に従って数をたくさん一度に作ることが出来ます。

```
Table[2^i, {i, 1, 10}]
```

{2, 4, 8, 16, 32, 64, 128, 256, 512, 1024}

```
Table[Sin[n/3 π], {n, 1, 10}]
```

$$\left\{\frac{\sqrt{3}}{2}, \frac{\sqrt{3}}{2}, 0, -\frac{\sqrt{3}}{2}, -\frac{\sqrt{3}}{2}, 0, \frac{\sqrt{3}}{2}, \frac{\sqrt{3}}{2}, 0, -\frac{\sqrt{3}}{2}\right\}$$

```
Table[Sin[θ], {θ, 0, π, π / 10}]
```

$$\left\{0, \frac{1}{4}\left(-1+\sqrt{5}\right), \sqrt{\frac{5}{8}-\frac{\sqrt{5}}{8}}, \frac{1}{4}\left(1+\sqrt{5}\right), \sqrt{\frac{5}{8}+\frac{\sqrt{5}}{8}}, 1,\right.$$
$$\left.\sqrt{\frac{5}{8}+\frac{\sqrt{5}}{8}}, \frac{1}{4}\left(1+\sqrt{5}\right), \sqrt{\frac{5}{8}-\frac{\sqrt{5}}{8}}, \frac{1}{4}\left(-1+\sqrt{5}\right), 0\right\}$$

記号「θ」は，「ESC+th+ESC」で入力できます。もしくは，パレットの

「BasicMathInput」ないしは「SpecialCharacters-J」からマウスで選択して入力します。

■ まとめと演習

ここまでの内容を降りかえって幾つかの計算をしてみましょう。

- 1から10までの和を計算し，その平均を求め小数に変換します。なお，平均を求める関数Meanを使う方法もあります。

```
Plus @@ Range[10]
```

```
55
```

```
Plus @@ Range[10] / 10
```

$$\frac{11}{2}$$

```
N[Plus @@ Range[10] / 10]
```

```
5.5
```

- 正弦（sin）の値を0からπまで$\frac{\pi}{10}$ごとに計算し，その和を求めてみます。

```
Table[Sin[θ], {θ, 0, π, π/10}]
```

$$\left\{0,\ \frac{1}{4}\left(-1+\sqrt{5}\right),\ \sqrt{\frac{5}{8}-\frac{\sqrt{5}}{8}},\ \frac{1}{4}\left(1+\sqrt{5}\right),\ \sqrt{\frac{5}{8}+\frac{\sqrt{5}}{8}},\ 1,\right.$$
$$\left.\sqrt{\frac{5}{8}+\frac{\sqrt{5}}{8}},\ \frac{1}{4}\left(1+\sqrt{5}\right),\ \sqrt{\frac{5}{8}-\frac{\sqrt{5}}{8}},\ \frac{1}{4}\left(-1+\sqrt{5}\right),\ 0\right\}$$

```
Plus @@ %
```

$$1+2\sqrt{\frac{5}{8}-\frac{\sqrt{5}}{8}}+2\sqrt{\frac{5}{8}+\frac{\sqrt{5}}{8}}+\frac{1}{2}\left(-1+\sqrt{5}\right)+\frac{1}{2}\left(1+\sqrt{5}\right)$$

```
FullSimplify[%]
```

$$1+\sqrt{5}+\sqrt{5+2\sqrt{5}}$$

Mathematica では，パーセント「%」を使うことで，直前に評価した計算結果を再利用することが出来ます。パーセント二個「%%」の場合は，2つ前に評価した結果の再利用になります。

また，関数SinはListableの属性を持っているので，次のようにRangeを使っても同じことを計算させることが出来ます。

```
FullSimplify[Plus @@ Sin[Range[0, Pi, Pi / 10]]]
```

$$1+\sqrt{5}+\sqrt{5+2\sqrt{5}}$$

演習

- サイコロを100回振ったときの目の合計と平均を求めましょう。

  Mathematica にはランダムな整数を作り出す関数RandomIntegerが組み込まれていますので，これを使ってください。例えば，一度のサイコロの試行を *Mathematica* にさせるには次のように入力します（評価するたびに値が変化します）。

```
RandomInteger[{1, 6}]
```

```
1
```

  なお，同じサイコロの出目に対して，合計と平均の両方を求めるときは，先に出目を求めておいて，「%」や「%%」を使って同じ出目を参照する必要があることに注意してください。

- 円周率を500桁求めましょう。

  厳密な数である「π」を近似数に変換することで求められますが，桁数も指定する必要があることに注意してください。

- 円の面積と，それを三角形分割して計算したときの値を比較しましょう。

半径を5として考えましょう。円の面積は当然次のように求まります。

```
π 5^2
```

```
25 π
```

二辺とその間の角から面積を求める公式を使って，円を4分割して面積を求めてみます。Tableの範囲指定で，最後のステップを取り除く必要があることに注意してください。

```
Plus @@ Table[1/2 Sqrt[5^2 * 5^2 - (5 * 5 * Cos[2 π/4])^2],
  {θ, 0, 2 π - 2 π/4, 2 π/4}]
```

```
50
```

これは，次のように変数を省略して書くことも出来ます。

```
Plus @@ Table[1/2 Sqrt[5^2 * 5^2 - (5 * 5 * Cos[2 π/4])^2], {4}]
```

```
50
```

分割数を増やして確認してみましょう。うまく，近似数への変換関数を使うことで，どの程度正しい値に近付いているか確認してください。

LECTURE02: 数学的な計算や操作の基本を覚えよう

Mathematica で理数系教材を作成するには，数学的な式の操作や様々な数式の表現方法を身に付ける必要があります。例えば，方程式を解いたり，行列の計算を行ったりすることを意味しています。そこで，この章の到達目標は，次のような計算を *Mathematica* にさせることになります。

- 円（$x^2 + y^2 = c$）と直線（$y = a\,x + b$）の交点を求めましょう。
- 掃き出し法で行列の階数（ランク）を求めてみましょう。
- 任意の関数の定積分を台形則で計算してみましょう。

■ 変数とその定義（*Mathematica* の記憶）

Mathematica では，すでに定義済みの関数などを除き，入力された文字はすべて変数（数学的な意味でも，プログラミングの意味でも）と見なされます。

```
a x + b
```

このとき， *Mathematica* が知っている（覚えている）変数や関数については，黒字で表示されますが，知らないものについては青字で表示されます。

```
{Plus, plus, A, I, e, E, Pi, pi}
```

Mathematica の組込み関数を入力したつもりなのに，黒字でなく青字になっていたら，何か入力ミスをしていることになります。上の例では，小文字のplus，大文字の*A*，小文字の*e*，小文字のpiについて， *Mathematica* は何も知らないことになります。

Mathematica の組込み関数の名前は，全て大文字から始まっています。既存の定義を上書きしたり，誤解を生まないよう，新しく使用する変数にはなるべく大文字を使わないようにしてください。

これらの変数に値を定義するには，等号一つ「=」を使用します。次のように変数*x*に値を定義すると， *Mathematica* の知っている変数になりますので，色が青字から黒字に変化します。

```
x = 1
```

```
1
```

```
y = 2
```

```
2
```

一度覚えさせた値は， *Mathematica* を再起動するまで（もしくは明示的に消去するまで）残ります。つまり，定義された変数を評価すると，定義されている値が結果として求まります。

```
x
```

```
1
```

```
3 x + y
```

```
5
```

定義した値をクリア（消去）するには，関数Clearを使用します。変数の痕跡を完全に削除するには，関数Removeを使用してください。

```
Clear[x]
```

Clearを用いて定義をクリアした後では，変数を評価しても， *Mathematica* はその値を知りませんので，変数をそのまま返してきます。値は返りません。

```
x
```

```
x
```

定義の参照は疑問符一つ「?」ないしは二つ「??」を使っても可能です。

```
?y
```

```
Global`y
```

y = 2

```
??y
```

```
Global`y
```

y = 2

これらの「?」と「??」を使うことで各関数のヘルプを見ることも出来ます。そのときアスタリスク「*」を使うことでアルファベットの「A」から始まる関数一覧なども得ることが出来ます（?? A*）。

演習

- 半径が3の円の面積を求めましょう。

  まずは円の面積Sを半径rを使って表現します。

```
S = π r^2
```

```
π r^2
```

  そして，半径を定義してから，もう一度面積を定義した変数Sを確認してみましょう。

多項式を操作してみよう

以下の操作をする前に，これまでに定義した変数をクリアしておきます。（理由は，値が定義されている変数は既に未知数ではないので，多項式の変数として使用できないからです）

```
Clear[x, y, z]
```

$x^2 + x + 1$という多項式を入力してみます。

```
x^2 + x + 1
```

$1 + x + x^2$

変数*x*は未知の文字ですから，入力した通りの多項式が出力されます。この多項式を*f*と定義しましょう。

```
f = x^2 + x + 1
```

$1 + x + x^2$

Mathematica では，多項式は降順でなく昇順で表示されます。降順で表示させたいときは，セルを選択し「SHIFT+CTRL+t」を入力します。もしくは，「セル」メニューから「形式変換」の中にある「TraditionalForm（慣用系）」を選択します。

もう一つ多項式を定義しましょう。なお，以下のように変数と係数の間のアスタリスク「*」や乗算を表すスペースは省略することが出来ます。

```
g = 3 x^2 + 3 x + 3
```

$3 + 3\,x + 3\,x^2$

数の演算と同じように，これらの多項式*f*と*g*に対しても，加減乗除が可能です。特に意識することなく，普通に計算させることが出来ます。

```
f + g
```

$4 + 4\,x + 4\,x^2$

```
f * g
```

$\left(1 + x + x^2\right)\left(3 + 3\,x + 3\,x^2\right)$

ここで，多項式*f*と*g*の積*f*×*g*は次のようにすることで，式を展開することが出来ます。一般に，*Mathematica* は明示的に命令で指定されない限り積を展開することはありません。

```
fg = Expand[f * g]
```

$$3 + 6\,x + 9\,x^2 + 6\,x^3 + 3\,x^4$$

関数Expandの逆操作は因数分解です。 *Mathematica* に因数分解を行わせる関数はFactorです。次のように因数分解された形が求まります。

```
Factor[fg]
```

$$3\,\left(1 + x + x^2\right)^2$$

Mathematica では，fgと間にスペースを書かずに入力したものは「fとgの積」にはなりません。単に，「fg」という名前の変数と見なされます。ぱっと見で良く似ているので間違いやすいので注意してください。

同じようにfをgで割ったものも，そのままでは簡単化されることはありません。

```
f / g
```

$$\frac{1 + x + x^2}{3 + 3\,x + 3\,x^2}$$

この簡単化には幾つか方法があります。一つは，既に出てきた関数Simplifyを使う方法です。

```
Simplify[f / g]
```

$$\frac{1}{3}$$

他にも，分数の約分を行う関数Cancelでも可能です。

```
Cancel[f / g]
```

$$\frac{1}{3}$$

演習

- 因数分解のほか，無平方分解，GCD計算なども *Mathematica* にさせることが出来ます。実際にやってみましょう。

  ドキュメントセンターで検索することで，対応する関数を調べられます。または，因数分解を行う関数のヘルプに関連する関数が載っています。

■ 方程式を解いてみよう（求根と代入操作）

Mathematica では等号二つ「==」で「両辺が数学的に等しい」ことを表現します。そして，等式の真偽値（正しいか正しくないか）が明らかなときは，等式を真偽値に置き換えて計算結果を返します。

```
1 == 1
```

```
True
```

```
1 == 2
```

```
False
```

等号一つ「=」と，等号二つ「==」はこのように意味が全く異なりますので，入力時には十分注意してください。なお，等しくないは「!=」と，感嘆符と等号を組み合わせて使います。

等号二つを使うことで， *Mathematica* 上で方程式を構成することが出来ます。例えば，次のように，二次方程式の一般形を表現することが出来ます。

```
a x^2 + b x + c == 0
```

この等式を指定した変数について解く（方程式を解く）には，関数Solveを使います。Solveは解きたい変数を指定できるので，*x*以外の変数についても簡単に解かせることが出来ます。

```
Solve[a x^2 + b x + c == 0, x]
```

$$\left\{\left\{x \to \frac{-b-\sqrt{b^2-4\,a\,c}}{2\,a}\right\},\ \left\{x \to \frac{-b+\sqrt{b^2-4\,a\,c}}{2\,a}\right\}\right\}$$

```
Solve[a x^2 + b x + c == 0, b]
```

$$\left\{\left\{b \to \frac{-c-a\,x^2}{x}\right\}\right\}$$

具体的な数値が求まる例を挙げてみます。

```
Solve[x^2 - 3 x - 10 == 0, x]
```

```
{{x → -2}, {x → 5}}
```

ここで，方程式を解いてくれる関数Solveの結果の形に注目してください。*Mathematica* では，この形式（→の左に変数が，右に数値など）で変数への代入を表しています。例えば，多項式「x^2-1」のxに3を代入するには次のようにします。

```
x^2 - 1 /. x → 3
```

```
8
```

記号「→」は，キーボードからハイフン「-」を入力し，大なり記号「>」を入力した結果である「->」と同じものであり，直後に何か入力すると，自動的に「->」が「→」に変換されます。

これは次のように書くことも出来ます。

```
x^2 - 1 /. Rule[x, 3]
```

```
8
```

```
ReplaceAll[x^2 - 1, x → 3]
```

```
8
```

```
ReplaceAll[x^2 - 1, Rule[x, 3]]
```

```
8
```

記号「/.」は「置換操作」に対応する関数ReplaceAllに，記号「–>」は「変換規則」と呼ばれる代入の内容を記すための関数Ruleに対応しています。

このように， *Mathematica* で方程式を解くと，その次に想定される代入操作を行いやすくなるように，結果が変換規則で得られます。従って，次のように解いた変数を置換することで，解のみからなるリストを得ることも可能です。

```
x /. Solve[a x^2 + b x + c == 0, x]
```

$$\left\{\frac{-b-\sqrt{b^2-4\,a\,c}}{2\,a},\ \frac{-b+\sqrt{b^2-4\,a\,c}}{2\,a}\right\}$$

演習

- 実際にSolveを使って，4次までの方程式を解いてみましょう。
- 実際にSolveを使って，5次以上の方程式を解いてみましょう。

  5次以上の方程式を解くと見たことのない結果が得られることがあります。それについては，次の項目で扱います。

代数的数と数値根（複雑な方程式を解く）

初等中等教育の立場から見ると，方程式を解く関数Solveは万能ではありません。数学的な制約（？）から一般に5次以上の方程式については解を具体的に求めることは出来ませんので，次のような結果が返されます。

```
Solve[x^5 - 3 x - 1 == 0, x]
```

```
{{x → Root[-1 - 3 #1 + #1^5 &, 1]},
 {x → Root[-1 - 3 #1 + #1^5 &, 2]},
 {x → Root[-1 - 3 #1 + #1^5 &, 3]},
 {x → Root[-1 - 3 #1 + #1^5 &, 4]},
 {x → Root[-1 - 3 #1 + #1^5 &, 5]}}
```

根として表示されているRootは， *Mathematica* において代数的数を表現するための関数です。正確に根を表現しようとすると，どうしてもこのように代数的数を使う必要が出てしまいます。

代数的数を表すRootオブジェクトの中で使用されているシャープ「#」やアパサンド「&」については，かなり後の章で出てくる純関数（Pure function）を表しています。

しかし，このままでは不便です。そこで， *Mathematica* には，厳密ではないけれども非常に理解しやすい数値として根を計算してくれる関数NSolveが用意されています。使い方は簡単で，Solveと全く同じです。

```
NSolve[x^5 - 3 x - 1 == 0, x]
```

```
{{x → -1.21465}, {x → -0.334734},
 {x → 0.0802951 - 1.32836 i},
 {x → 0.0802951 + 1.32836 i}, {x → 1.38879}}
```

このように数値的に近似された根を求めた場合，次のように代入しても完全に0にはなりません。しかし，十分0に近い数値となっています。

```
x^5 - 3 x - 1 /. NSolve[x^5 - 3 x - 1 == 0, x]
```

```
{1.77636 × 10^-15, 0., -8.88178 × 10^-16 - 4.44089 × 10^-16 i,
 -8.88178 × 10^-16 + 4.44089 × 10^-16 i, 8.88178 × 10^-16}
```

なお，NSolveを直接使うのではなく，Rootオブジェクトに数値化を行う関数*N*を使っても，近似値を得ることが出来ます。

```
N[Solve[x^5 - 3 x - 1 == 0, x]]
```

```
{{x → -1.21465}, {x → -0.334734}, {x → 1.38879},
 {x → 0.0802951 - 1.32836 i}, {x → 0.0802951 + 1.32836 i}}
```

演習

- 実際に方程式を解いてみましょう。
  - $x^2 + 3x + 2 = 0$
  - $x^6 - 2x^2 = 3x - 1$
- 関連する次の関数について調べて使ってみましょう。
  - FindRoot
  - Series

調べるときは，選択して「F1」キーを押します。

多項式の並び替えや係数の取り出し

通常，次のような多項式を入力した場合には昇順に並び替えられ表示されます。

```
x y z + z y + x + y + z + 1
```

```
1 + x + y + z + y z + x y z
```

これを*x*や*y*に関して個別にまとめたい場合は，次のように関数Collectを使います。

```
Collect[%, x]
```

```
1 + y + z + y z + x (1 + y z)
```

```
Collect[%, y]
```

```
1 + x + z + y (1 + z + x z)
```

```
Collect[%, z]
```

```
1 + x + y + (1 + y + x y) z
```

また，係数だけを取り出したい場合には次のように関数CoefficientListを使います。これは指定した変数xに関する定数項と一次の項の係数からなるリストになっています。

```
CoefficientList[%, x]
```

```
{1 + y + z + y z, 1 + y z}
```

そろそろ出てくる関数名も長くなってきました。*Mathematica* では途中まで関数を入力し，「CTRL+k」を押すとある程度関数名を補完してくれます。特に，長い名前の関数などを入力する場合には重宝しますので，ぜひ使ってみてください。

ベクトルと行列の計算

３次元ベクトルとも考えられる，次のようなリストa_1，a_2，a_3を考えます。

```
a1 = {1, 2, 3};
a2 = {4, 5, 6};
a3 = {7, 8, 9};
```

Mathematica では行末にセミコロン「;」を付けることで，その命令によって出力されるはずの出力の表示を抑制することが出来ます。更に上記の様に，３つの定義を同時に行うことも可能です。このときセミコロン「;」の付いた行の結果は表示されませんが，きちんと代入動作が行われています。

実際に評価して，定義されているか確認してみます。

```
a1
```

```
{1, 2, 3}
```

リストに四則演算を行うと，ベクトルのスカラー倍に似た効果となります。

```
3 * a1
```
```
{3, 6, 9}
```

```
a1 / 3
```
$$\left\{\frac{1}{3},\ \frac{2}{3},\ 1\right\}$$

```
a1 + 1
```
```
{2, 3, 4}
```

リスト同士の四則演算は，各要素毎に四則演算が行われます。従って，サイズの異なるリスト同士の四則演算を行うことは出来ません。

```
a1 + a2
```
```
{5, 7, 9}
```

```
a1 * a2
```
```
{4, 10, 18}
```

リストをベクトルに見立て，その内積を計算する場合にはドット「.」を使います。結果はスカラーとなります。

```
a1 . a2
```
```
32
```

この３つのベクトルから成る行列$A=(a_1,\ a_2,\ a_3)^t$は次のように表現できます。

```
A = {a1, a2, a3}
```
```
{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}
```

行列を数学の慣用表現で出力するには関数MatrixFormを使います。

```
MatrixForm[A]
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

ここで，MatrixFormは行列出力用の関数です。他にもFormが付く関数は多数ありますので，「?*Form」などを実行してみると良いでしょう。

ここからは行列について考えてみましょう。まずは，次のような行列を定義します。なお，大文字の変数名は *Mathematica* の組込みの関数や定数と重複する可能性があるので，本来はなるべく避けた方が賢明です。

```
A = {{1, 0, 0}, {0, 2, 0}, {0, 3, 0}};
B = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

リストのリストが行列であると考えれば，再帰的に同じ演算が繰り返されるだけで，基本的にリストの演算と変わりはありません。従って，数との四則演算や行列同士の四則演算についても，同じ規則が成立します。

```
3 * A
```

```
{{3, 0, 0}, {0, 6, 0}, {0, 9, 0}}
```

```
A + B
```

```
{{2, 2, 3}, {4, 7, 6}, {7, 11, 9}}
```

```
A * B
```

```
{{1, 0, 0}, {0, 10, 0}, {0, 24, 0}}
```

通常の行列の積を計算するには，ドット「.」を使います。

```
A.B
```

```
{{1, 2, 3}, {8, 10, 12}, {12, 15, 18}}
```

「*」と「.」で全く意味が違います。行列の演算の時には「.」を使うことを忘れないようにしてください。

演習

- 実際にベクトルや行列の演算を行ってみましょう。

  サイズが異なるリスト同士の場合など，様々なケースを想定して実験してみてください。行列の慣用表現での確認（MatrixForm）もしてみてください。

- 行列に関する様々な操作も用意されています。

  ドキュメントセンターで，転置，単位行列，逆行列などを検索して実際に使ってみましょう。

リストや行列の部分操作

Mathematica のリストには，プログラミング言語での配列に似た操作を行うことも可能です。これにより，行列の各要素を数学の慣用表現に近い形で参照したり変更したりすることが可能となります。

まずは，行列*A*を次のように定義します。

```
MatrixForm[A = {{1, 2, 3}, {2, 0, -3}, {1, 1, -1}}]
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 0 & -3 \\ 1 & 1 & -1 \end{pmatrix}$$

この行列の(2, 2)成分と(3, 1)成分を取り出すには次のようにします。つまり，*A*[[*i*, *j*]]は行列Aの(*i*, *j*)要素を意味します。

```
A[[2, 2]]
```

```
0
```

```
A[[3, 1]]
```

```
1
```

これの応用として行列Aの(i, j)要素の値だけを上書きすることが出来ます。

```
A[[2, 2]] = a
```

```
a
```

```
MatrixForm[A]
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & a & -3 \\ 1 & 1 & -1 \end{pmatrix}$$

また，単に行列の行のみ，または，列のみを取り出したい場合は，次のように列番号ないしは行番号を「All」とします。

```
A[[2, All]]
```

```
{2, a, -3}
```

```
A[[All, 2]]
```

```
{2, a, 1}
```

行の取り出しの場合は，行列ではなく単なるリストのi番目を取り出すという考え方もでき，その場合は，次のように「All」を省略することも可能です。

```
A[[2]]
```

```
{2, a, -3}
```

従って，次のように2行目全体を入れ換えることも可能です。

```
A[[2]] = Range[1, 3]
```

```
{1, 2, 3}
```

```
MatrixForm[A]
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 1 & -1 \end{pmatrix}$$

まとめと演習

ここまでの内容を降りかえって幾つかの演習をしてみましょう。

演習

- 円（$x^2 + y^2 = c$）と直線（$y = a\,x + b$）の交点を求めましょう。

  複数の変数，複数の方程式（連立方程式）を扱う場合も，*Mathematica* への命令はSolveやNSolveをそのまま使えます。ただし，複数の変数や方程式を与えるときは，それをリスト（集合）として中括弧で取りまとめて指定する必要があります。

- 掃き出し法で行列の階数（ランク）を求めてみましょう。

  例えば，行列*A*の(2, 1)要素を1行目を使って消去するには，次のような操作を行います。結果を，再度，行列*A*に代入しているので，順次，行列が変化していくことになります。

```
A[[2]] = A[[2]] - A[[1]] * (A[[2, 1]] / A[[1, 1]])
```

```
{0, 0, 0}
```

```
MatrixForm[A]
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & 0 & 0 \\ 1 & 1 & -1 \end{pmatrix}$$

- 任意の関数の定積分を台形則で計算してみましょう。

  台形則とは，積分区間を小区間に分け，その小区間毎に台形の面積を求め，その和によって定積分の値を求める手法です。Tableと代入（/.）をうまく組み合わせることで，各小区間での台形の面積を求め，その和を計算して定積分を求めることになります。

LECTURE03: グラフィックスと動的操作の基本

教材でも研究用途でも，最終的に得られた数式やデータをグラフィックスに可視化することで，その事象を理解し易くすることが出来ます。この章の到達目標は，次のようなグラフィックスや動的コンテンツを *Mathematica* に作成させることになります。

- 与えられた不等式を満たす領域を描画する。
- 曲線の方程式と与えられた点における接線を合わせて描画する。
- 極座標形式の関数を回転させるアニメーションを作成する。

とりあえずグラフィックスを作ってみよう

*Mathematica*のグラフィックス機能はとても多彩です。グラフィックス関連の関数は以下に示すようにたくさんあります。まずは，二次元の関数グラフに関するものを調べてみます。

```
?*Plot
```

▼ System`

ArrayPlot	ListLinePlot	LogPlot
ContourPlot	ListLogLinearPlot	MatrixPlot
DateListPlot	ListLogLogPlot	ParametricPlot
DensityPlot	ListLogPlot	Plot
GraphPlot	ListPlot	PolarPlot
LayeredGraphPlot	ListPolarPlot	RegionPlot
ListContourPlot	LogLinearPlot	ReliefPlot
ListDensityPlot	LogLogPlot	TreePlot

次に，三次元の関数のグラフに関するものを調べてみます。

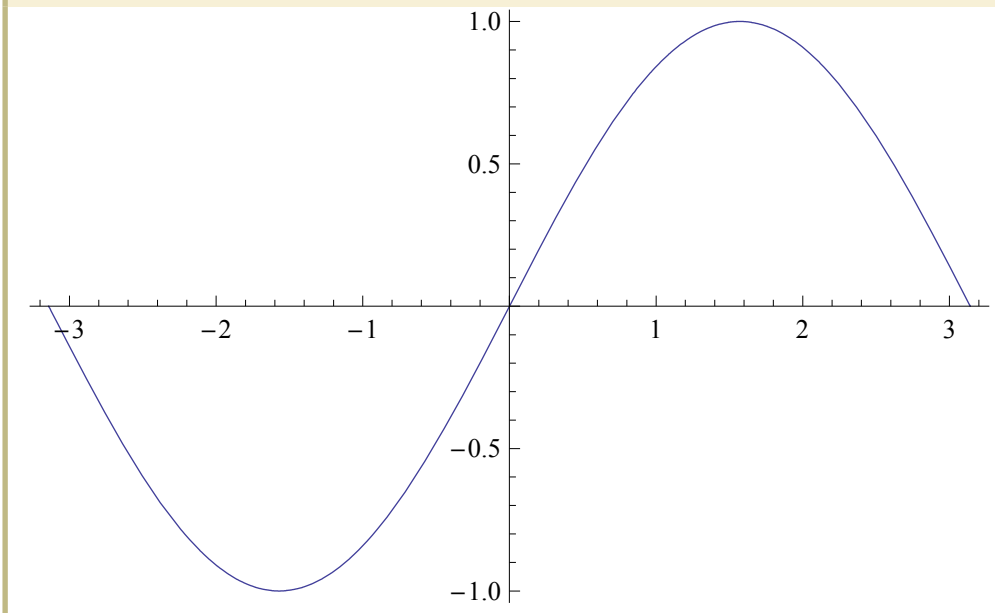
```
?*Plot3D
```

▼ System`

ContourPlot3D	ListPointPlot3D	RegionPlot3D
GraphPlot3D	ListSurfacePlot3D	RevolutionPlot3D
ListContourPlot3D	ParametricPlot3D	SphericalPlot3D
ListPlot3D	Plot3D	

まずは，簡単な三角関数のグラフを$-\pi$からπまでプロット（描画）してみます。

```
Plot[Sin[x], {x, -Pi, Pi}]
```

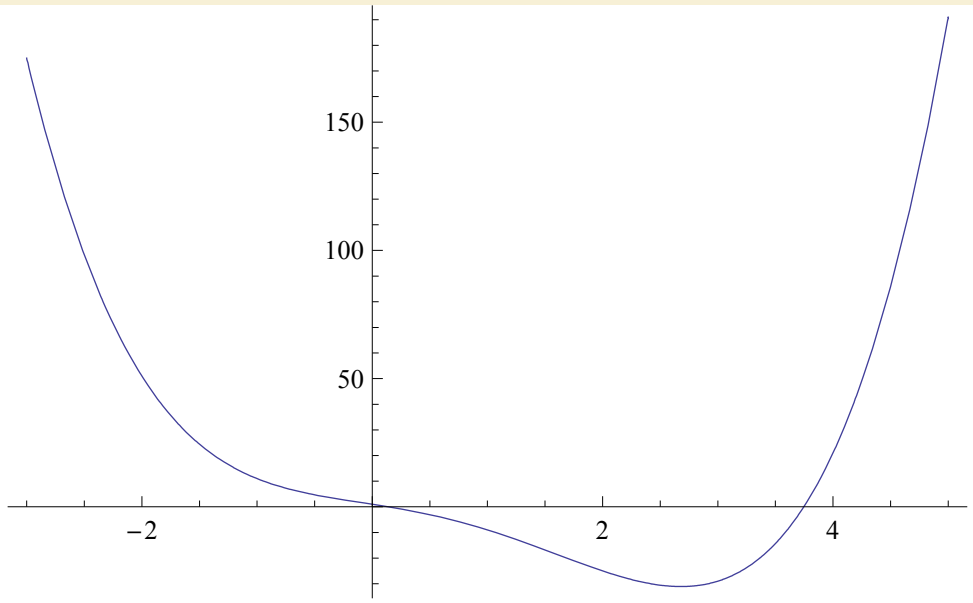


基本的に陽関数のグラフは，以下のような形で命令します。

関数名[曲線の関数，{変数，始点，終点}]

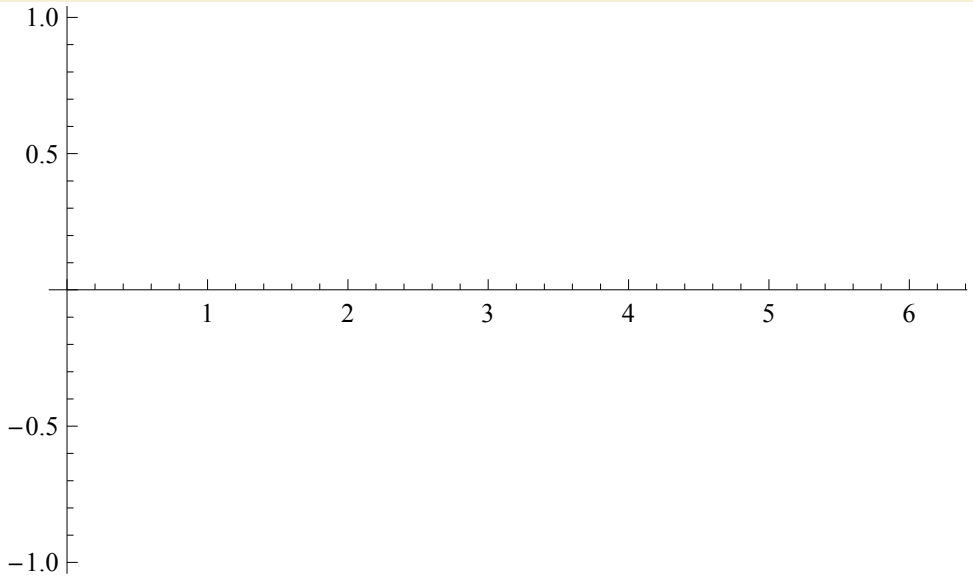
例えば４次関数のグラフは次のようになります。

```
Plot[x^4 - 3 x^3 - x^2 - 7 x + 1, {x, -3, 5}]
```



なお，曲線の関数に未知のパラメータを含んでいると，数直線上のどこに点や線を描けば良いのか *Mathematica* は判断できませんので，次のように真っ白なグラフが描かれることになります。

```
Plot[a x^2 + b x + c, {x, 0, 2 π}]
```

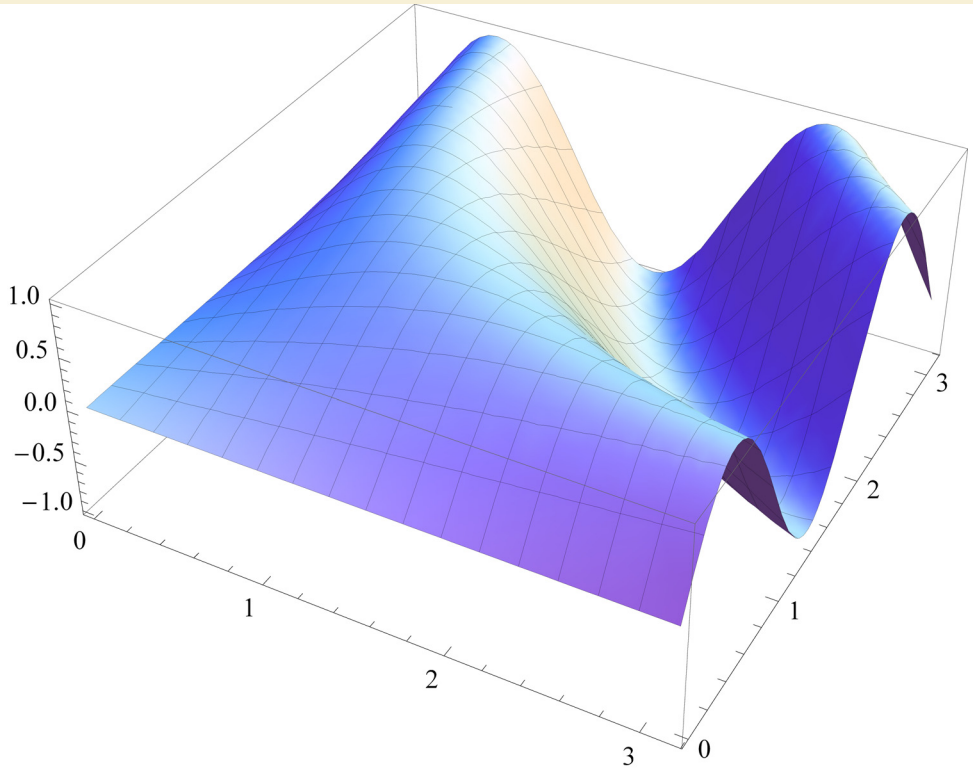


二次元のグラフィックスには，グラフィックスメニューから「描画ツール」や「グラフィックスインスペクタ」を開くことで，レタッチするこ

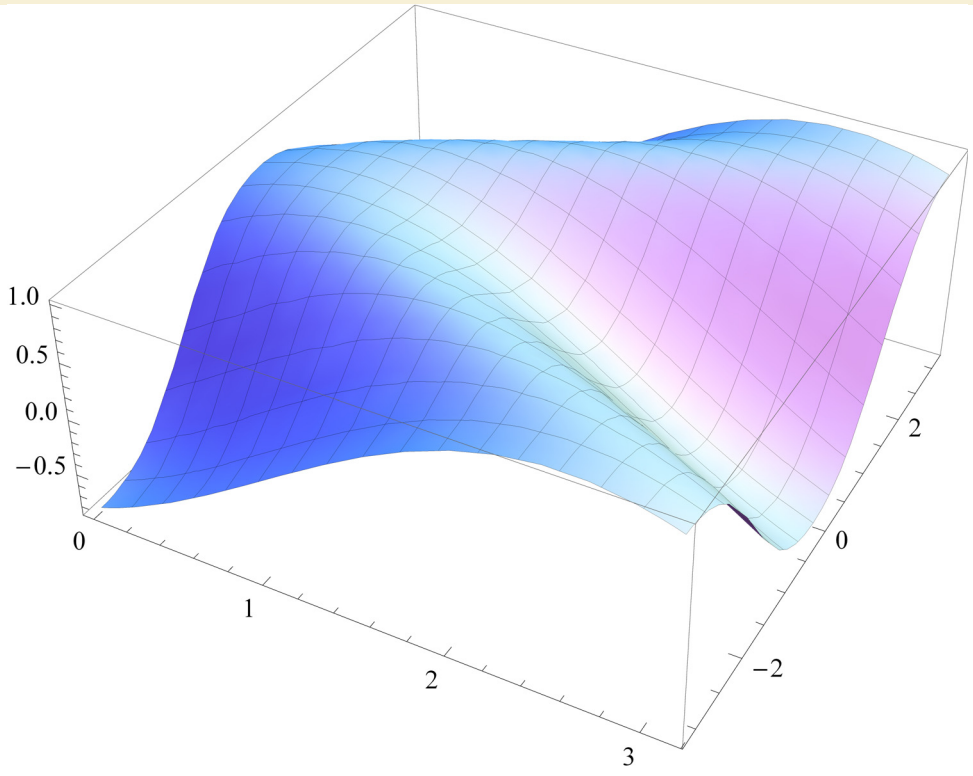
とや線などの色を簡単に変更することも可能です。

同様に三次元のグラフは次のようになります。範囲指定が増えるだけです。

```
Plot3D[Sin[x y], {x, 0, π}, {y, 0, π}]
```



```
Plot3D[Sin[x + Cos[y]], {x, 0, π}, {y, -π, π}]
```



三次元のグラフィックスは，マウスでドラッグすると回転させることも出来ます。CTRLキーやSHIFTキーを押しながらドラッグすると異なる操作も可能です。

演習

- パラメータ表示の関数（媒介変数表示の関数）のグラフを描きましょう。

  ドキュメントセンターで「パラメータ表示」を検索します。

- 等高線図や密度図を描きましょう。

  ドキュメントセンターで「等高線」を検索します。

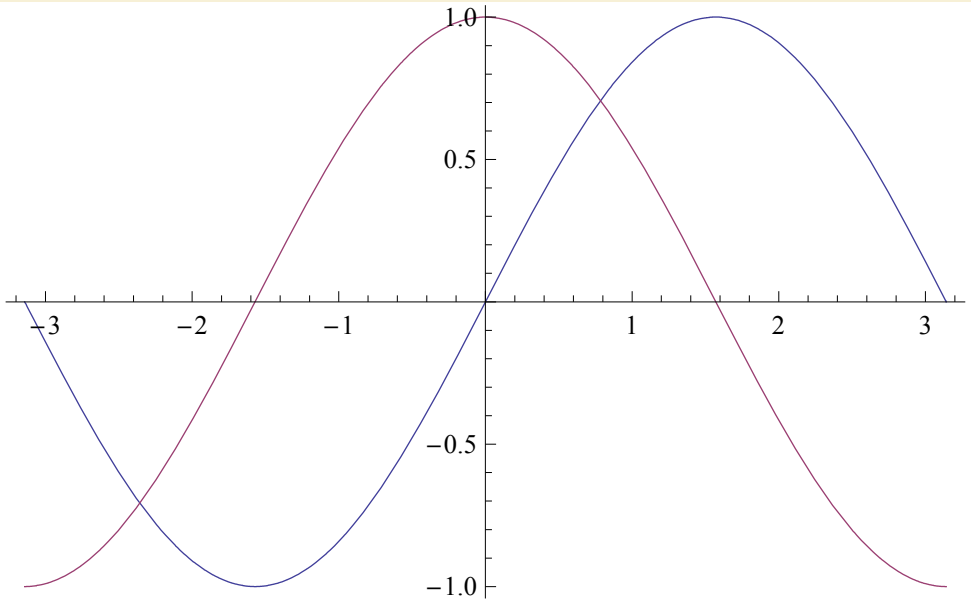
- データ列を可視化してみましょう。

  ドキュメントセンターで「データの可視化」を検索します。

■ 複数のグラフィックスを重ねて表示する

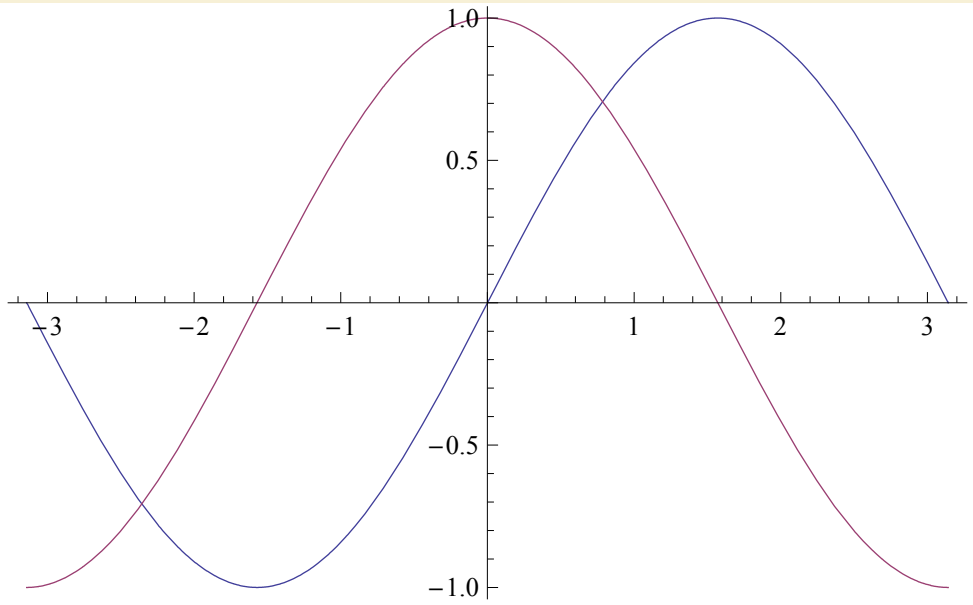
正弦と余弦の関数を同時に描画したい場合も，単純に二つの関数（SinとCos）をリストで指定するだけです。三つ以上の関数を同時に描画したい場合も同じようにリストで指定することで，いくつでも同時に描くことが出来ます。

```
Plot[{Sin[x], Cos[x]}, {x, -Pi, Pi}]
```



しかし，複数のグラフを同時に描画させると，どの線がどの関数であるかわからなくなります。そのような場合，関数Tooltipを活用します。次のように，関数のリストの前にアットマーク「@」を使って前置形で指定することで，描画されている線の上にマウスを持っていくと，対応する関数がツールチップとしてポップアップして表示されます。

```
Plot[Tooltip@{Sin[x], Cos[x]}, {x, -Pi, Pi}]
```

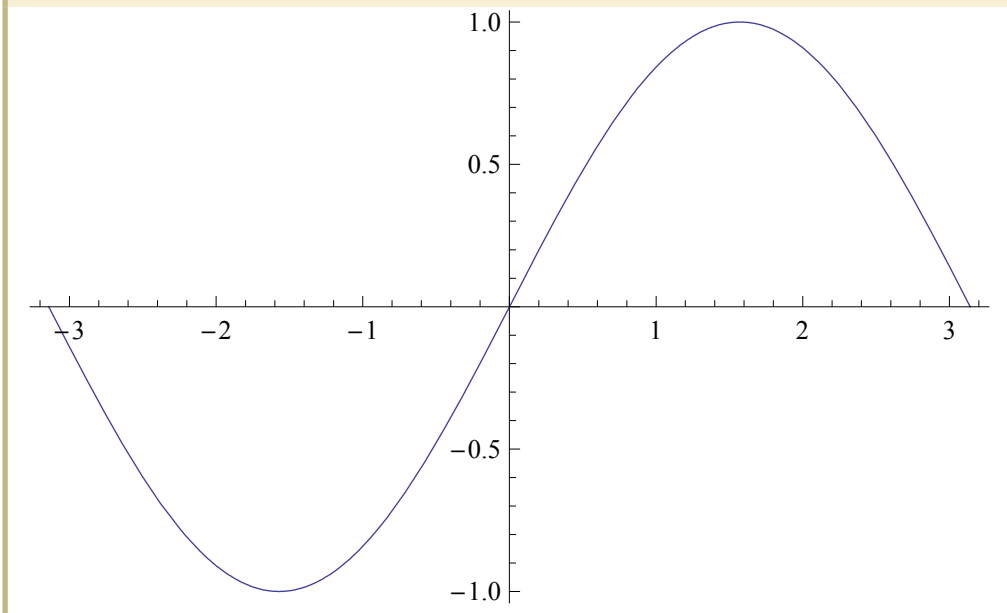


グラフィックスを変数に代入しておき，後で再表示することも出来ます。*Mathematica* ではあらゆるものを変数に代入することが可能となっています。なお，グラフィックスについても行末にセミコロン「;」を付けることで，出力（この場合はグラフ）の表示を抑制できます。

```
g1 = Plot[Sin[x], {x, -π, π}];
g2 = Plot[Cos[x], {x, -π, π}];
```

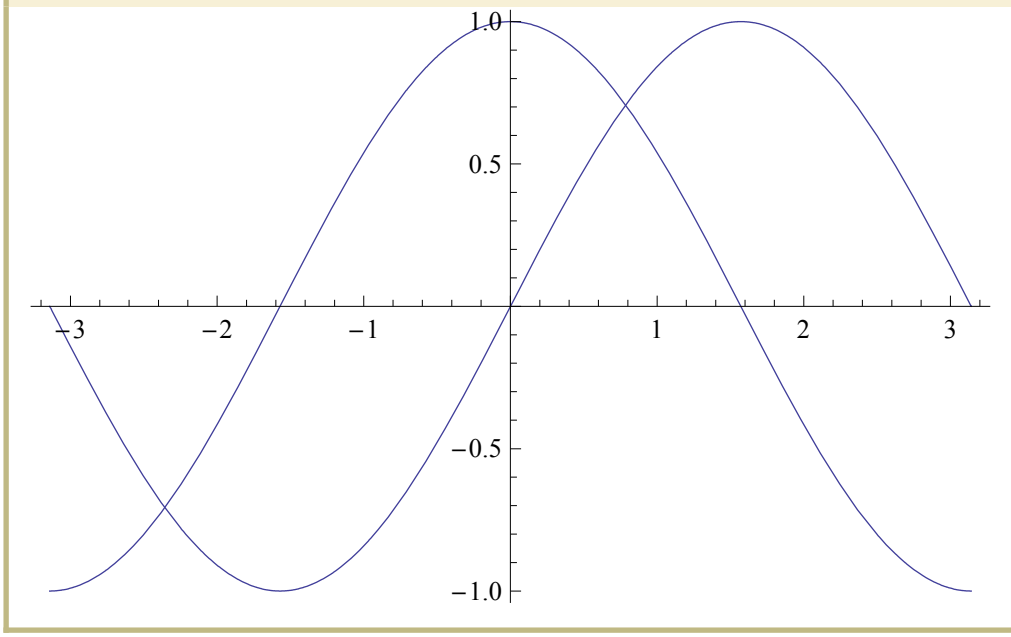
変数を評価すると代入されているグラフが表示されます。

```
g1
```



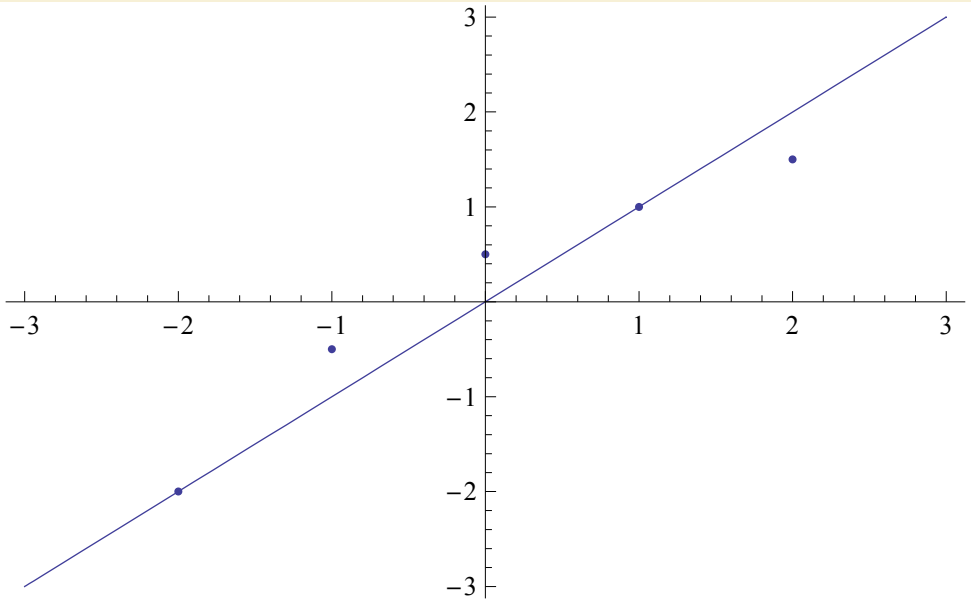
このように変数に保存されているグラフィックスなどを，改めて重ねて表示したい場合，関数Showを利用します。

```
Show[g1, g2]
```



この関数Showを利用することで，データ（点列）のグラフと滑らかな関数のグラフを重ねて表示して，ずれなどを視覚的に確認することも出来ます。

```
Show[
 Plot[x, {x, -3, 3}],
 ListPlot[{{-3, -3.5}, {-2, -2}, {-1, -0.5},
   {0, 0.5}, {1, 1}, {2, 1.5}, {3, 3.5}}]
]
```



■ 描画方法を細かく指示する（オプションの使い方）

Mathematica には各命令の挙動（グラフィックスであれば装飾など）を細かく指定するためにオプションと呼ばれる仕組みが備わっています。例えば，関数Optionsを使うことで，グラフを描く関数Plotに備わるオプションの全てを確認することが出来ます。

Options[Plot]

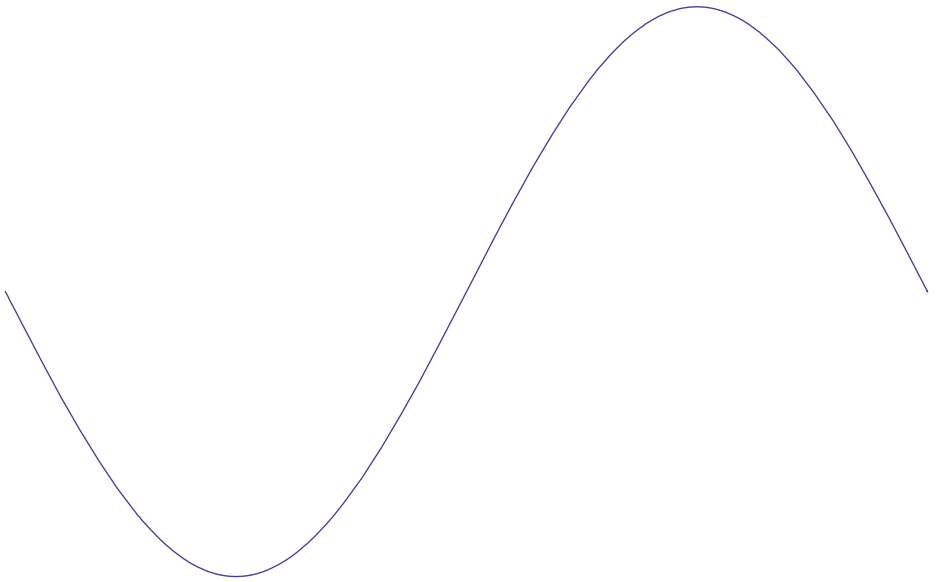
```
{AlignmentPoint → Center, AspectRatio → 1/GoldenRatio,
 Axes → True, AxesLabel → None, AxesOrigin → Automatic,
 AxesStyle → {}, Background → None,
 BaselinePosition → Automatic, BaseStyle → {},
 ClippingStyle → None, ColorFunction → Automatic,
 ColorFunctionScaling → True, ColorOutput → Automatic,
 ContentSelectable → Automatic,
 DisplayFunction :→ $DisplayFunction, Epilog → {},
 Evaluated → Automatic, EvaluationMonitor → None,
 Exclusions → Automatic, ExclusionsStyle → None,
 Filling → None, FillingStyle → Automatic,
 FormatType :→ TraditionalForm, Frame → False,
 FrameLabel → None, FrameStyle → {},
 FrameTicks → Automatic, FrameTicksStyle → {},
 GridLines → None, GridLinesStyle → {},
 ImageMargins → 0., ImagePadding → All,
 ImageSize → Automatic, LabelStyle → {},
 MaxRecursion → Automatic, Mesh → None,
 MeshFunctions → {#1 &}, MeshShading → None,
 MeshStyle → Automatic, Method → Automatic,
 PerformanceGoal :→ $PerformanceGoal, PlotLabel → None,
 PlotPoints → Automatic, PlotRange → {Full, Automatic},
 PlotRangeClipping → True,
 PlotRangePadding → Automatic,
 PlotRegion → Automatic, PlotStyle → Automatic,
 PreserveImageOptions → Automatic, Prolog → {},
 RegionFunction → (True &), RotateLabel → True,
 Ticks → Automatic, TicksStyle → {},
 WorkingPrecision → MachinePrecision}
```

これらのオプションは，変換規則で与えられており，関数Optionsで表示される変換規則の右辺値（置き換え先）が，標準で利用される設定を表しています。例えば，「Axes → True」は「座標軸を描くか」を指定するオプションAxesの標準設定がTrue（真）であることを意味しています。実際，関数Plotに何もオプションを指定しないで使うと，座標軸は標準で描かれています。

関数名[普通の引数，オプション，...]

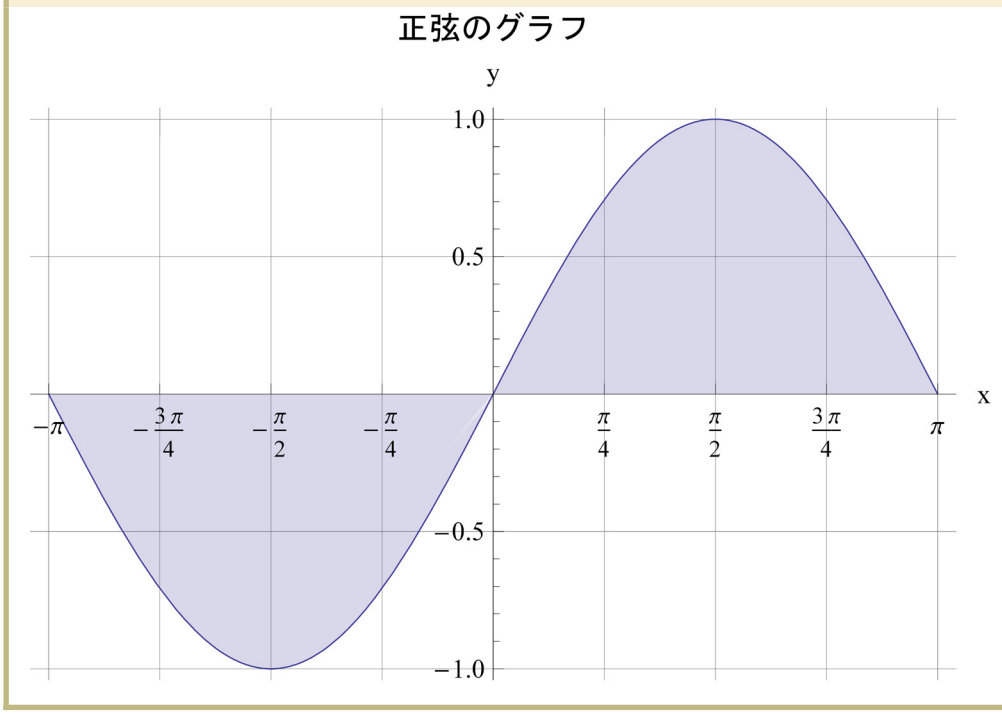
これらのオプションは，命令の最後にカンマ「,」区切りで追加して指定します。例えば，座標軸を描かないようにするには，次のようにオプションを指定します。なお，変換規則のところでも出てきましたが，右矢印「→」は，ハイフン「–」に続いて大なり「>」を入力することで自動的に変換されます。

```
Plot[Sin[x], {x, -π, π}, Axes → False]
```



これらのオプションを組み合わせて使うことで，次のようなグラフを描くことも可能です。

```
Plot[Sin[x], {x, -Pi, Pi}, Filling → Axis,
 GridLines → {Range[-Pi, Pi, Pi / 4], Automatic},
 Ticks → {Range[-Pi, Pi, Pi / 4], Automatic},
 PlotLabel → "正弦のグラフ", AxesLabel → {"x", "y"}]
```



各オプションは，オプション名を選択してから，「F1」キーを押すことでドキュメントセンターから検索されて表示されます。

演習

- *Mathematica* が描くグラフは，縦横比が異なります。これを変更するオプション「AspectRatio」を使ってみましょう。

  縦横比を同じにするには，「AspectRatio → Automatic」を指定します。

- *Mathematica* が描くグラフは，その描画範囲が自動的に決定されます。これを変更するオプション「PlotRange」を使ってみましょう。

  横軸を[−5, 5]の範囲で，縦軸を[−2, 2]の範囲で描くには，「PlotRange → {{−5, 5}, {−2, 2}}」を指定します。

- *Mathematica* が描く線や点の色などは自動的に決められています。これを変更するオプション「PlotStyle」を使ってみましょう。

破線でグラフを描かせるには，「PlotStyle → Dashed」を指定します。なお，リストで複数の関数を指定した場合は，PlotStyleにも同じ要素数のリストを指定することで，それぞれの関数の描画スタイルを指定することが可能です。

■ 表計算ソフトにあるようなグラフを描いてみる

Mathematica で棒グラフや円グラフを描くには，パッケージと呼ばれる拡張機能を利用します。パッケージは自分で作ることや購入することも出来ますが，ほとんどの場合， *Mathematica* に最初から付属しているパッケージを利用するだけで済みます。

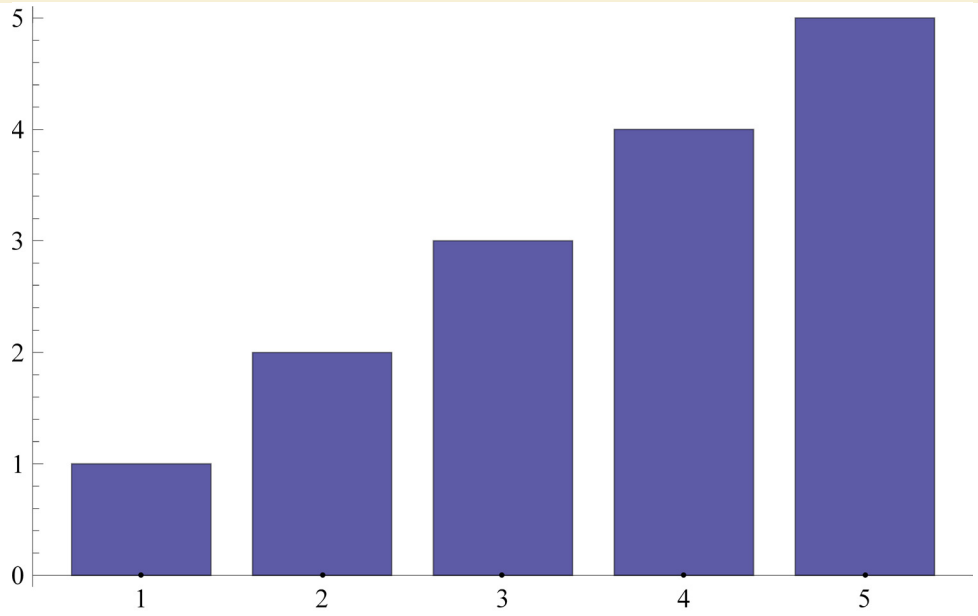
パッケージを利用するには，関数Needsを使います。例えば，棒グラフを描くには，パッケージ「BarCharts`」を次のようにして読み込みます。

```
Needs["BarCharts`"]
```

パッケージに含まれる関数を使う前に，必ずパッケージを読み込んでください。この手順が前後しますと，パッケージに含まれる関数を利用できないこともあります。

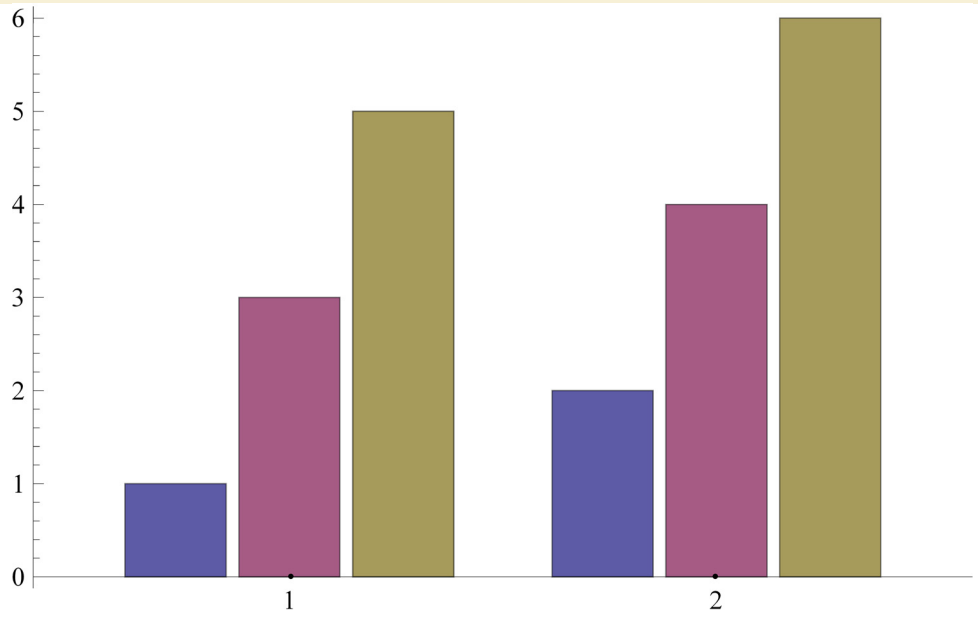
棒グラフを作成するには，関数BarChartを利用します。実際のデータをリストとして引数に指定することで，次のように棒グラフを作成することが出来ます。

```
BarChart[{1, 2, 3, 4, 5}]
```



引数としてリストのリストを与えると，内側のリストが同じ種類のデータを表現していると認識され，以下のように色分けが行われます。

```
BarChart[{{1, 2}, {3, 4}, {5, 6}}]
```

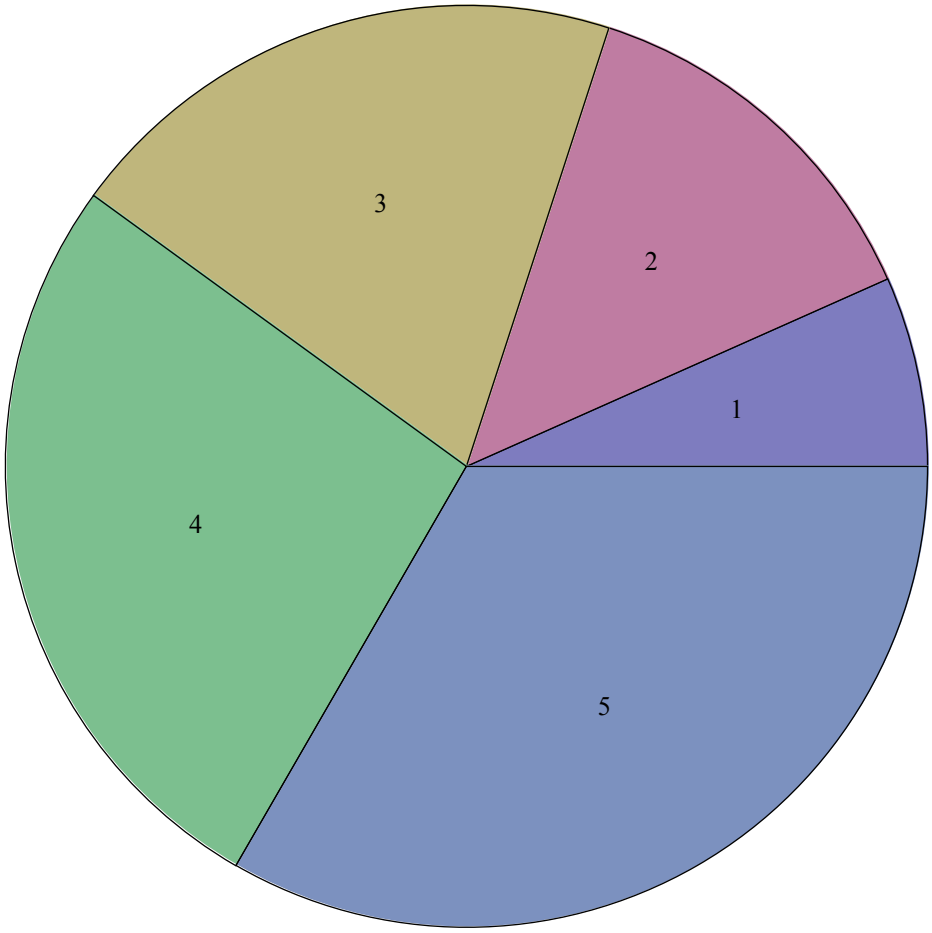


同様に，円グラフを描くには，パッケージ「PieCharts`」を次のようにして読み込みます。

```
Needs["PieCharts`"]
```

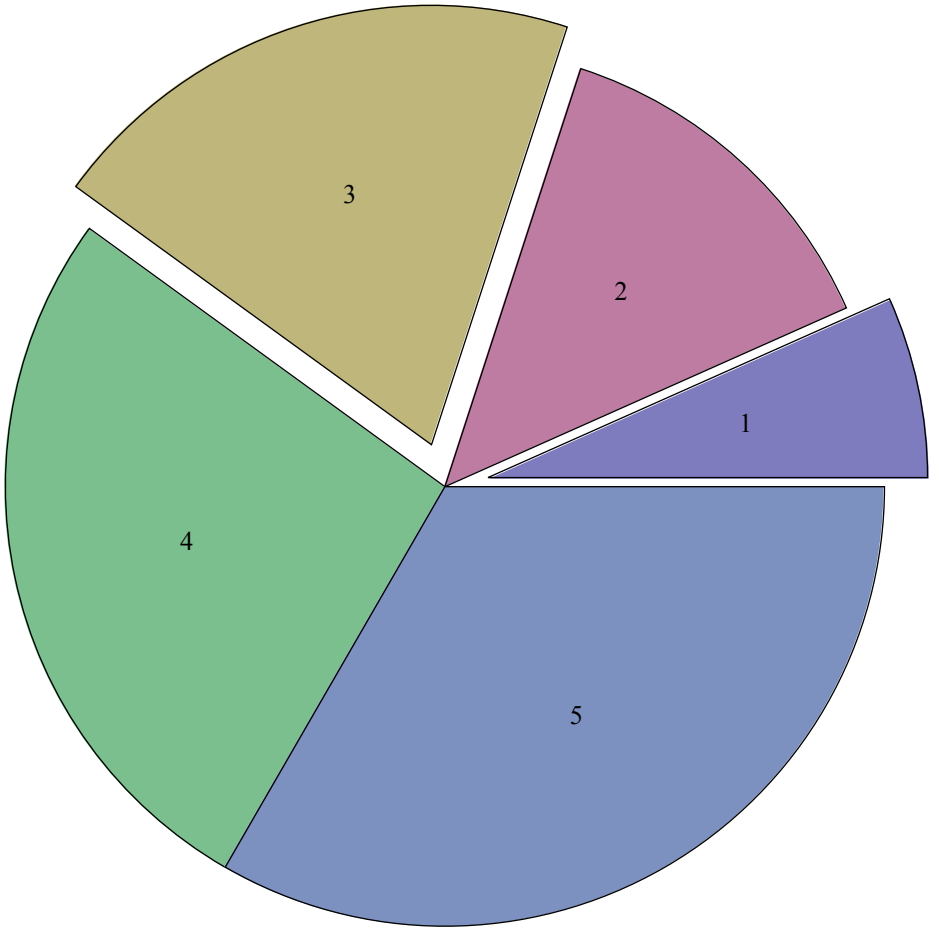
利用方法は基本的に棒グラフと同じです。

```
PieChart[{1, 2, 3, 4, 5}]
```



これらのパッケージ関数にもオプションが用意されており，他の関数と同じくOptionsを使って確認できます。以下は，オプションを指定して，円グラフの一部を切り離したものです。

```
PieChart[{1, 2, 3, 4, 5}, PieExploded → {1, 3}]
```

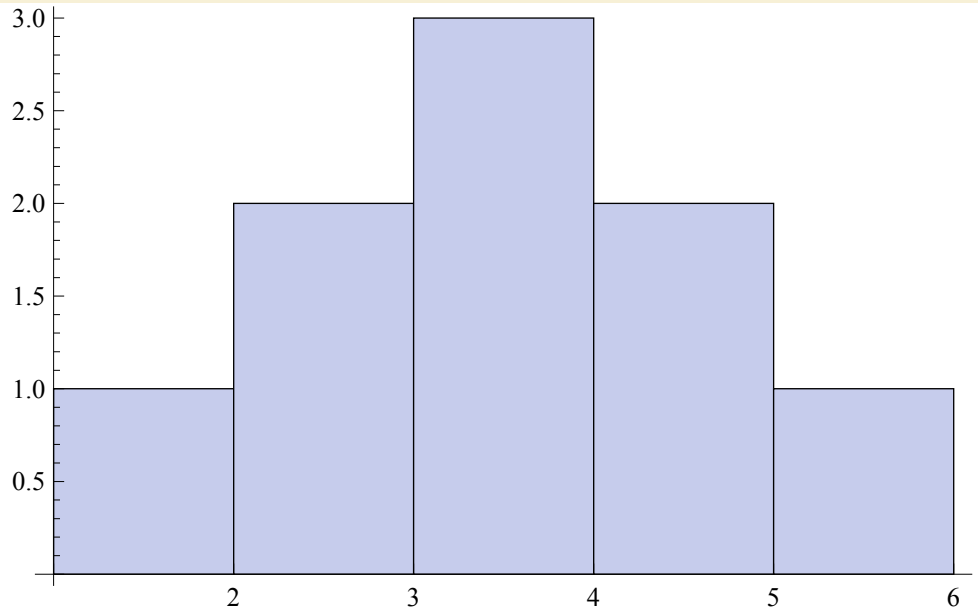


棒グラフに似ていますが，パッケージ「Histograms`」を次のようにして読み込むことで，ヒストグラムをデータから直接描かせることが可能になります。

```
Needs["Histograms`"]
```

指定の方法は棒グラフと同じですが，度数を自動的に計算して棒グラフとして表示してくれます。

```
Histogram[{1, 2, 2, 3, 3, 3, 4, 4, 5}]
```



演習

- 棒グラフのパッケージには，紹介した以外にも次の関数も含まれています。実際に使ってみましょう。

  PercentileBarChart，StackedBarChart，BarChart3D

- ヒストグラムのパッケージには，紹介した以外にも次の関数も含まれています。実際に使ってみましょう。

  Histogram3D

- ベクトル場プロットパッケージを利用すると，磁力線などの様子を簡単に可視化することが出来ます。実際に利用してみましょう。

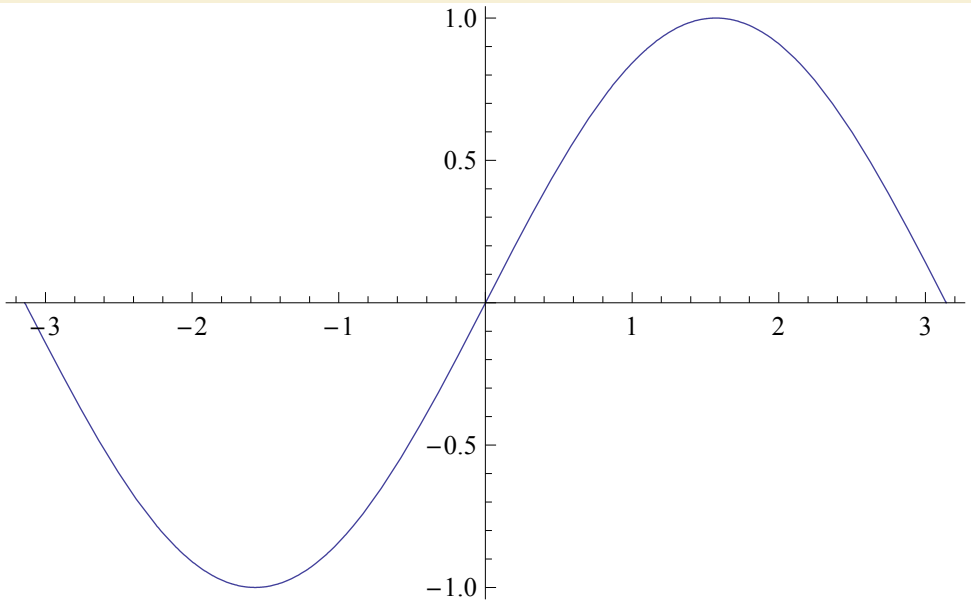
  ドキュメントセンターで「ベクトル場プロットパッケージ」を検索します。

アニメーションを作ってみよう

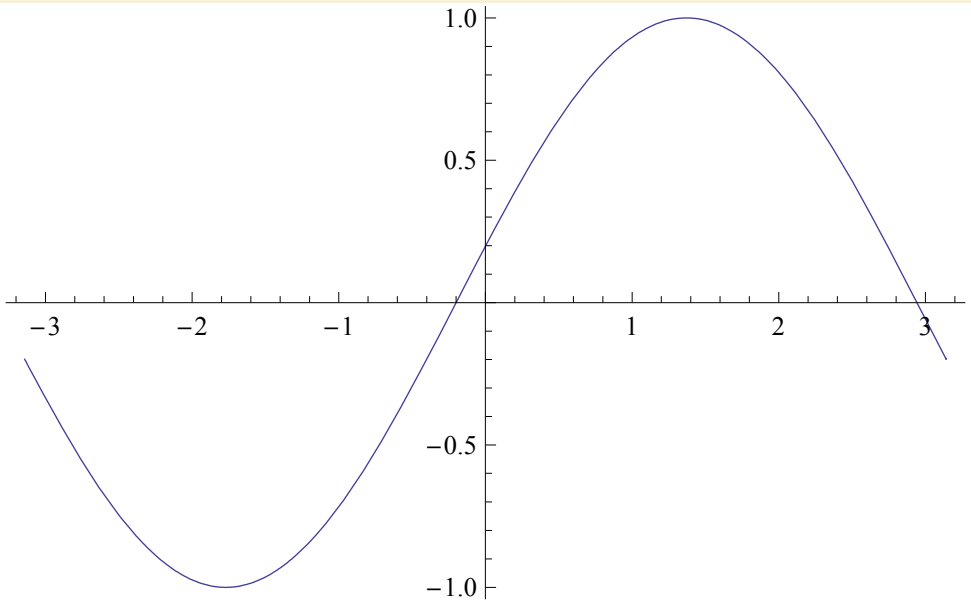
Mathematica では非常に簡単に動的に変化するグラフ（アニメーション）を作成することが可能です。以下では，正弦のグラフをアニメーションさせる手順を最初から説明していきます。

まず，正弦のグラフの初期位相を少しずつ変えて描画してみましょう。

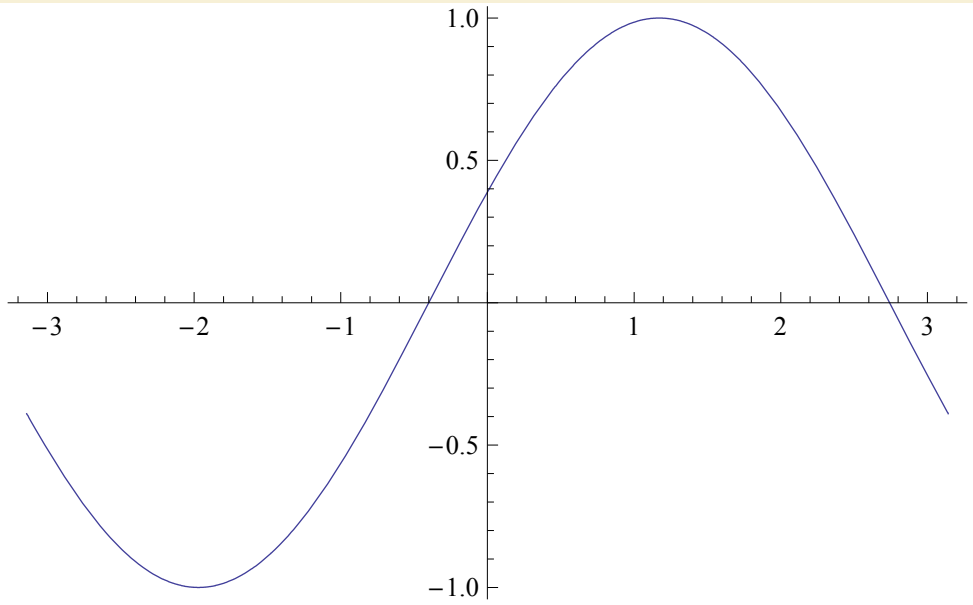
```
Plot[Sin[x + 0.0], {x, -Pi, Pi}]
```




```
Plot[Sin[x + 0.2], {x, -Pi, Pi}]
```



```
Plot[Sin[x +0.4], {x, -Pi, Pi}]
```



赤字の部分が変化している（変化させたい）部分になります。この変化している数字を関数Rangeで作成すると次のようになります。

```
Range[0.0, 0.4, 0.2]
```

```
{0., 0.2, 0.4}
```

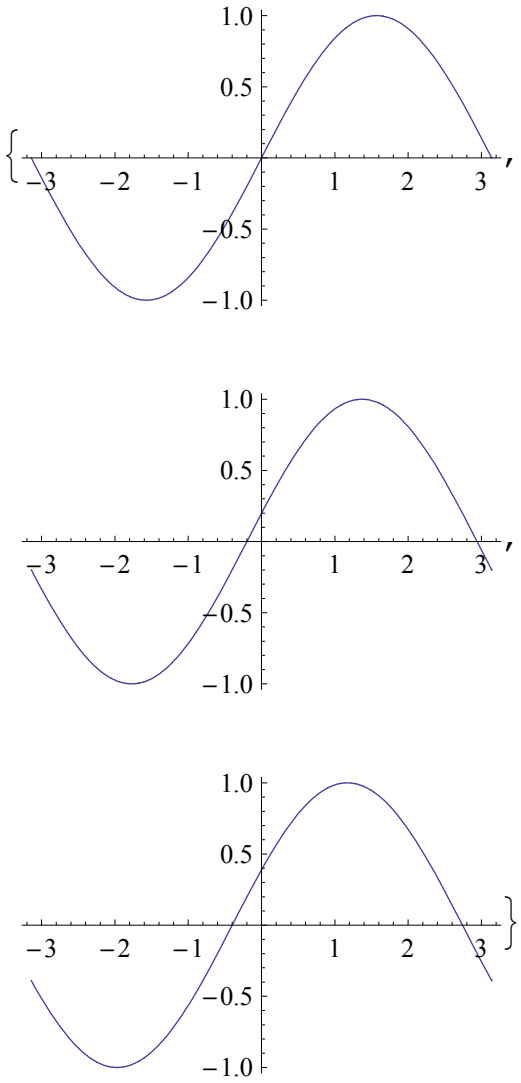
同じリストを関数Tableで作成すると次のようになります。以前に，リストの作成方法のところで出てきた方法と同じです。

```
Table[i, {i, 0, 0.4, 0.2}]
```

```
{0., 0.2, 0.4}
```

いま，変化している部分だけをリストとして作成する方法を確認しました。残りの部分は変化しないので，どのグラフにおいても共通しています。そこで，上のTable関数の第一引数に変化していない部分を組み込んでしまいしょう。次のように，一度に複数のグラフを作成することが出来ます。

```
Table[Plot[Sin[x +i], {x, -Pi, Pi}], {i, 0, 0.4, 0.2}]
```

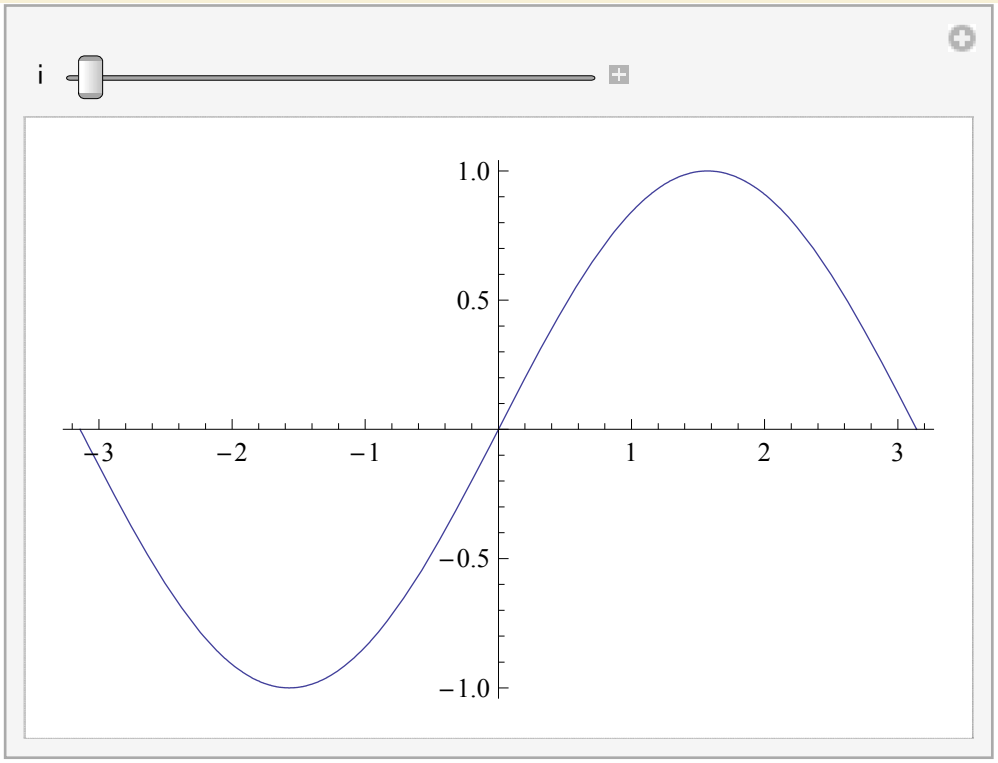


Mathematica のバージョン5.2までは，このようにTable関数などを使って，複数のグラフをリストとして作成することで，パラパラ漫画のようなアニメーションを作成出来ました。*Mathematica* のバージョン6からは，グラフィックスもただのデータとして扱われるようになったため，単なるリストとして表示されます。

次に，この状態で，Tableの代わりに関数Manipulateを使ってみましょう。Manipulateは，バージョン6からの新しい関数で，教材作成を行う際の非常に強力な機能を提供してくれます。

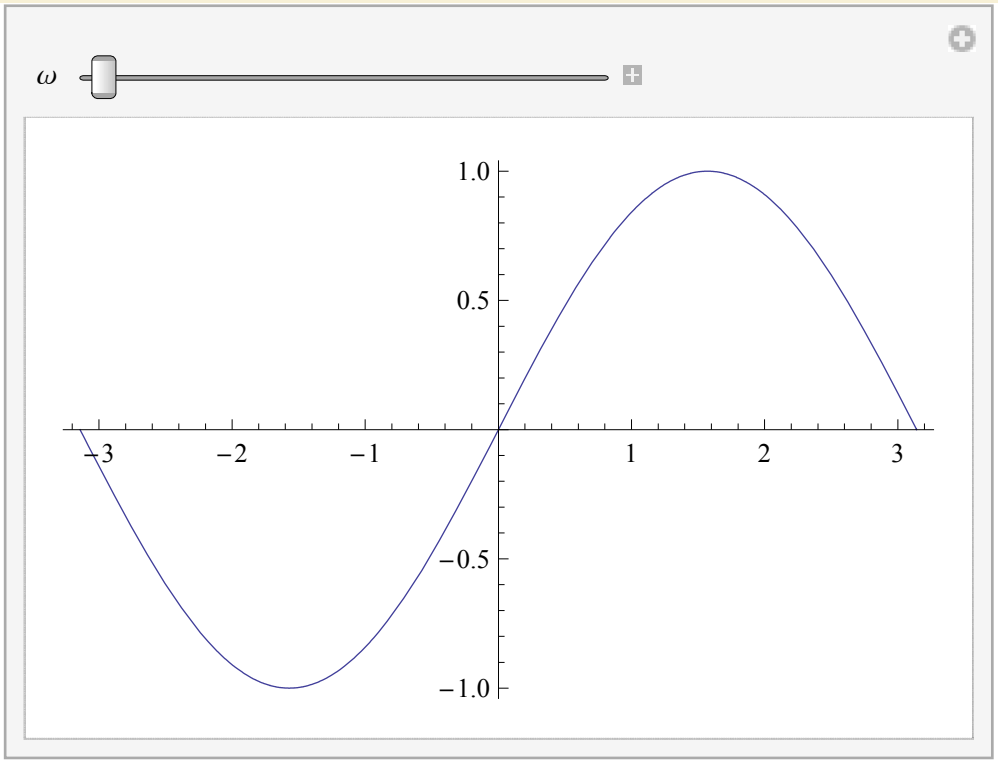
Manipulateは，グラフィックスのリストを作る代りに，スライダーをマウスで動かすことにより，各状態を切替えて表示してくれるパネルを生成します。また，スライダーの横のプラス記号をクリックすることで，より詳細な動作も行えます。

```
Manipulate[Plot[Sin[x +i], {x, -Pi, Pi}],
 {i, 0, 0.4, 0.2}]
```



下記のパネルはプラス記号を開いた状態です。スライダーの動きを離散的にする必要がなければ，範囲指定を「{ω, 0, 2π}}のように，ステップ幅を省略して指定してください。

```
Manipulate[Plot[Sin[x + ω], {x, -π, π}], {ω, 0, 2 π}]
```



演習

- 初期位相を動かすアニメーションとして，正弦と余弦のグラフを同時に描画してみましょう。

  まずは，Plotを使って初期位相を固定したものを作成し，変化させたいところを変数に置き換え，その変数の動く範囲をManipulateで指定するようにしましょう。

- Manipulateで線の太さが変化するアニメーションを作成しましょう。

  Manipulateではあらゆるものを変化させることが出来ます。PlotStyleオプションと，線の太さを指定するThicknessを組み合わせれば，スライダーを動かすと線の太さが変化するアニメーションを作成できます。

アニメーションをFlash動画として保存しよう

Manipulateで作成したアニメーションはFlash動画として保存することが可能です。これを行うには関数Exportを使います。方法は非常に単純で，次のように，第一引数に保存先のファイル名（拡張子を「.swf」にする）を，第二引数に出力したいManipulate命令全体を指定するだけです。

Export["ファイル名.swf", Manipulate[中身は省略]]

```
Export["sin.swf",
 Manipulate[Plot[Sin[x + θ], {x, -Pi, Pi}],
  {θ, -Pi, Pi}]]
```

```
sin.swf
```

Flash動画への保存は時間がかかることがあります。また，特に指定をしなければ，Windowsの場合，ファイルはマイドキュメントに作成されます。

まとめと演習

ここまでの内容を降りかえって幾つかの演習をしてみましょう。

演習

- 与えられた不等式を満たす領域を描画する。

  不等式をの領域を描画するには，RegionPlotを利用します。また，状況によっては，オプションのRegionFunctionをPlotなどと組み合わせて利用します。

- 曲線の方程式と与えられた点における接線を合わせて描画する。

  接線を求めるには導関数を計算する必要があります。偏微分は「D」で，全微分は「Dt」で計算することが出来ます。例えば，$f(x) = x^2 - 1$の導関数は次のように求められます。

```
D[x^2 - 1, x]
```

```
2 x
```

従って，$x = -1$における接線の傾きは次のように置換と変換規則を用いて

求めることが出来ます。

```
D[x^2 - 1, x] /. x → -1
```

```
-2
```

すなわち，$x = -1$における接線の式$l(x)$は，次のように求めることが出来ます。

```
(D[x^2 - 1, x] /. x → -1) (x - (-1)) + (x^2 - 1 /. x → -1)
```

```
-2 (1 + x)
```

- 極座標形式の関数を回転させるアニメーションを作成する。

  三角関数とParametricPlotを使うと簡単に極座標形式の関数を描画したり，それを回転させることが出来ます。場合によっては，関数PolarPlotを利用する方が良いこともあります。

LECTURE04: プログラミングでちょっと難しいこと

Mathematica は様々な数学関数などが使えるプログラミング言語として考えることも出来ます。この章の到達目標は，そのプログラミングの基礎を学び，次のような多少複雑なことを *Mathematica* に実行させることです。

- Euclidの互除法で整数の最大公約数を求めてみる。
- 行列の行の基本変形をしてくれる関数を作る。
- 関数を与えると「接線が動くアニメーション」を作成する関数を作る。

■ 遅延的な定義（遅延評価）を理解しよう

変数に値を代入することについて，もう一度，復習してみましょう。次のように等号「=」を使うことで，変数*r*に値「10」を定義することが出来ました。

```
r = 10
```

```
10
```

一旦定義された値を *Mathematica* は覚えているので，再度，変数を評価すると覚えている値が出力されます。疑問符「?」を使うことで，*Mathematica* が覚えている実際の内容を確認することも出来ます。

```
r
```

```
10
```

```
? r
```

```
Global`r
```

```
r = 10
```

では，次に乱数を発生する関数RandomRealを使うことを考えます。この関数は，次の例のように，評価するたびに0から1までの実数の疑似乱数を返します。

```
RandomReal[]
```

```
0.068253
```

この関数を使って，次のように変数*r*に値を定義します。

```
r = RandomReal[]
```

```
0.254714
```

このとき，変数*r*の定義は次のようになっています。上の定義では，等号の右辺にRandomRealが存在しているのに，実際に覚えられている内容は数値「0.254714」だけです。

```
?r
```

```
Global`r

r = 0.254714
```

等号を使って変数の定義を行うとき， *Mathematica* は「右辺の評価」を行い，「その結果を左辺に割り当てる」ことをします。従って，RandomRealで生成された実数が変数*r*に定義されることになります。このように，右辺が即時に評価されるタイプの定義のことを，*Mathematica* では「即時的な定義（即時評価）」と言います。

一方，次のようにコロンと等号「:=」を使って，変数の定義を行うことも可能です。

```
r := 10
```

```
?r
```

```
Global`r

r := 10
```

コロンと等号による定義の場合も，変数を再度評価すると， *Mathematica* が覚えている値「10」を返します。

```
r
```

```
10
```

しかし，次のように右辺にRandomRealを持ってくると状況が変化します。

```
r := RandomReal[]
```

先ほどとは異なり，変数*r*を評価するたびに値が変化します。

```
r
```

```
0.939255
```

```
r
```

```
0.960808
```

このとき，変数*r*の定義を確認すると，先ほどとは異なり，右辺にRandomRealが残ったまま定義されています。変数*r*が呼び出されると，この定義に基づき，右辺のRandomRealが実行され，その結果，実数の疑似乱数が出力されます。

```
?r
```

```
Global`r
```

```
r := RandomReal[]
```

つまり，コロンと等号を使って変数の定義を行うとき，*Mathematica* は「右辺の評価」を行わず，「そのまま左辺に割り当てる」ことをします。従って，右辺の関数RandomRealが実行されずにそのまま残るので，左辺が呼び出されたときに毎回乱数が生成されることになります。このように，右辺が後で評価されるタイプの定義のことを，*Mathematica* では「遅延的な定義（遅延評価）」と言います。

	定義方法	左辺の値	用途
即時評価	等号 「=」	右辺の結果が代入	常に変化しない定数など
遅延評価	コロンと等号 「:=」	右辺の式そのもの	変化の可能性のあるもの

演習

- 評価する度にランダムに天気予報する変数を定義してみましょう。

まず，リストを使って天気予報の内容を準備します。 *Mathematica* では，二重引用符で囲んだ文章を文字列として認識します。

```
{"晴れ", "曇り", "雨", "晴れのち曇り", "曇り時々雨"}
```

```
{晴れ, 曇り, 雨, 晴れのち曇り, 曇り時々雨}
```

RandomChoiceを使うことで，リストの中から一つをランダムに選ぶことが出来ます。

```
RandomChoice[{"晴れ", "曇り", "雨", "晴れのち曇り",
  "曇り時々雨"}]
```

```
曇り
```

評価するたびに結果が変わる可能性があるので，これを変数に定義する場合，コロンと等号を使った遅延的な定義にする必要があります。実際に，変数tenkiを定義してみます。

```
tenki :=
 RandomChoice[{"晴れ", "曇り", "雨", "晴れのち曇り",
   "曇り時々雨"}]
```

そうすると，変数tenkiを評価するたびに，ランダムな予報が得られます。

```
tenki
```

```
晴れのち曇り
```

```
tenki
```

```
雨
```

パターンマッチとユーザー定義関数

遅延的な定義を使って，次の性質を持つ関数myzeroを作ってみましょう。

```
myzero (n) = 0
```

Mathematica の関数を使うときに，引数を大括弧「[」と「]」で囲んで使っているのを思い出してください。従って，上の数学的な定義の一部をそのまま，即時的な定義を使って書き直すと次のようになります。

```
myzero[0] = 0;
myzero[1] = 0;
myzero[n] = 0;
```

実際に，いくつか計算をさせてみましょう。

```
myzero[0]
```

```
0
```

```
myzero[1]
```

```
0
```

```
myzero[5]
```

```
myzero[5]
```

myzeroc[5]に対して， *Mathematica* は計算をせずにそのままの式を返してきました。なぜでしょうか。念のため，*Mathematica* の覚えている関数myzerocの定義の内容を確認してみます。

```
?myzero
```

```
Global`myzero
```

```
myzero[0] = 0
```

```
myzero[1] = 0
```

```
myzero[n] = 0
```

myzero[5]を評価するとき， *Mathematica* はこの定義を上から順番に調べ，myzero[5]と同じものがあるかを調べます。myzero[0]やmyzero[1]であれば，同じものがあるので，その定義に基づいて右辺の値を返します。しかし，myzero[5]と同じものはないので，仕方なくそのままの式を返してしまうのです。従って，以下のように「myzero[*n*]」を評価すると確かに右辺の式が得られます。

```
myzero[n]
```

```
0
```

しかしながら，全ての*n*に対して，個別に定義を全て書いていくことは現実的ではありません。そのため， *Mathematica* では「パターンマッチ」という仕組みを活用して，この問題を解決します。とりあえず，先ほど定義した内容を消去しておきます。関数Clearを使うことで，定義を消去出来ます。

```
Clear[myzero]
```

Mathematica には「どんなものにもマッチ」するワイルドカードのような記号が用意されています。それは，アンダースコアひとつ「_」です。従って，次のように定義することで，どのような引数が指定されてもマッチする左辺を作り出すことが出来ます。

```
myzero[_] = 0
```

```
0
```

念のため，定義の内容を確認してみます。

```
?myzero
```

```
Global`myzero
```

```
myzero[_] = 0
```

どこにも具体的なmyzero[0]やmyzero[5]の定義はありませんが，次のように実際に評価すると，どんなものにもマッチするアンダースコアひとつ「_」に「0」や「5」がマッチし，そのときの右辺値「0」が返されます。

```
myzero[0]
```

```
0
```

```
myzero[5]
```

```
0
```

以下の説明に従って，実際に評価を行う前に，「ALT+.」（Altキーとドットキーを同時に押すこと）により計算の強制終了（評価の放棄）が行えることを確認して下さい。意図的に強制終了が必要な間違った定義を行ってるので，注意してください。

アンダースコアひとつ「_」を使って，次の性質を持つフィボナッチ数列を計算する関数を作ってみましょう。なお，*Mathematica* はFibonacciというフィボナッチ数列を計算するための関数を持っていますが，練習のため自作します。

```
fib (1) = 1, fib (2) = 2,
fib (n) = fib (n – 1) + fib (n – 2)  (n = 3, 4, …)
```

なお，具体的な「*n*」の値によって右辺の結果が変化する可能性のある「fib[*n*]」は，即時的な定義ではなく，遅延的な定義をしなければいけないことに気がつかれると思います。即ち，ここまでの知識を総動員すると，フィボナッチ数列を計算する関数fibの定義は次のように行うことになります。

```
fib[1] = 1;
fib[2] = 1;
fib[_] := fib[n – 1] + fib[n – 2];
```

実際に，いくつか計算をさせてみましょう。

```
fib[1]
```

```
1
```

```
fib[2]
```

```
1
```

```
fib[3]
```

```
$RecursionLimit::reclim : 最大再帰回数256を超えています．  ≫
$RecursionLimit::reclim : 最大再帰回数256を超えています．  ≫
$RecursionLimit::reclim : 最大再帰回数256を超えています．  ≫
General::stop :
  計算中，$RecursionLimit::reclimのこれ以上の出力は表示されません．  ≫
$Aborted
```

fib[3]に対して， *Mathematica* は「最大再帰回数256を超えています」というエラーを出します。理由は後述しますが，この時点では強制終了（評価の放棄）をするしかありません。もう一度，定義を確認してみましょう。

```
? fib
```

```
Global`fib

fib[1] = 1

fib[2] = 1

fib[_] := fib[n - 1] + fib[n - 2]
```

fib[3]は最初の二つとはマッチしないので，最後の式にマッチすることになります。従って，fib[3]は次の式に展開されます。

```
fib[n - 1] + fib[n - 2]
```

Mathematica は更に定義に従って，fib[$n-1$]とfib[$n-2$]を計算しようとしま

す。即ち，fib[$n-1$]もfib[$n-2$]も定義の最後のfib[_]にマッチするので，それぞれが次の式に展開されます。

```
fib[n - 1] + fib[n - 2]
```

そして，*Mathematica* は更に定義に従って，，，，と無限ループに陥ります。延々と繰り返しているうちに *Mathematica* は何かおかしいぞと気がついて，256回も繰り返しているというエラーを表示してきます。これが，fib[3]に対してエラーが表示される理由です。

まとめると，次の定義式において，左辺のどんなものにもマッチするアンダースコアひとつ「_」と，右辺の「n」がそれぞれ別のものとして扱われているために，意図した通りに動かないことになります。

```
fib[_] := fib[n - 1] + fib[n - 2]
```

解決策は簡単です。左辺のどんなものにもマッチするアンダースコアひとつ「_」に便宜的に名前を付け，それを右辺で「n」の代りに使うだけです。実は，アンダースコアひとつ「_」の左側に好きな名前を付けられるようになっているので，「n_」と名前を付けるだけで，右辺の「n」と関連づけられることになります。従って，正しいフィボナッチ数列の定義は次式になります。

```
Clear[fib]
```

```
fib[1] = 1;
fib[2] = 1;
fib[n_] := fib[n - 1] + fib[n - 2];
```

左辺のパターンと右辺の変数が正しく関連付けられているとき，変数の文字色が「緑色」になります。青色のままの場合，それは関連付けられていない変数を意味します。

実際に計算してみましょう。

```
fib[5]
```

```
5
```

```
fib[6]
```

```
8
```

```
Table[fib[i], {i, 1, 6}]
```

```
{1, 1, 2, 3, 5, 8}
```

補注

再帰を行う関数を定義する場合，以下のように遅延的な定義の中で，即時的な定義を再帰的に行っていくように記述します。このようにしないと，計算速度が遅すぎて使いものになりません。

```
fib[1] = 1;
fib[2] = 1;
fib[n_] := (fib[n] = fib[n - 1] + fib[n - 2]);
```

実際に呼び出されたときに，即時定義が行われていくため，fib[8]を評価する前と後で，その定義内容が変化します。これによって，余計な再帰を発生させずに済みます。

```
? fib
```

```
Global`fib

fib[1] = 1

fib[2] = 1

fib[n_] := fib[n] = fib[n - 1] + fib[n - 2]
```

```
fib[8]
```

```
21
```

```
? fib
```

```
Global`fib
```

```
fib[1] = 1

fib[2] = 1

fib[3] = 2

fib[4] = 3

fib[5] = 5

fib[6] = 8

fib[7] = 13

fib[8] = 21

fib[n_] := fib[n] = fib[n - 1] + fib[n - 2]
```

ループ処理と純関数（Pure Function）

今度は，与えられた整数を割り切る最大素数を求める関数を定義してみます。素因数分解を行う関数FactorIntegerを利用して，素因数の中でもっとも大きいもの（最後の要素の最初）を取り出しています。

```
largestprime[n_] := FactorInteger[n][[-1, 1]]
```

実際に使ってみると次のように，最大素数が求まります。

```
largestprime[2 * 3 * 7]
```

```
7
```

しかし，このままでは整数を複数個含むリストを引数として指定すると，期待する正しい答え（この場合は，{3, 2, 5}）は求まりません。

```
largestprime[{3, 4, 5}]
```

```
{5, 1}
```

なぜ，このような結果になるかというと，どのようなものにもマッチする「n_」は，リスト全体にマッチしており，それがそのままFactorIntegerに引き渡されるからです。そのため，こちらの期待とは異なり，次の命令が実行されてしまいます。

```
FactorInteger[{3, 4, 5}][[-1, 1]]
```

では，どのようにしたら，次のように展開出来るのか考えていきます。

```
{largestprime[3], largestprime[4], largestprime[5]}
```

Mathematica には，与えられたリストの各要素に同じ関数を適用した結果のリストを求める関数Mapが用意されています。これを使うことで，次のように，リストの各要素にlargestprimeを適用した結果を得られます。

```
Map[largestprime, {3, 4, 5}]
```

```
{3, 2, 5}
```

Mapは四則演算と同じく省略して書くことが出来るようになっています。省略記号は，スラッシュアットマーク「/@」です。従って，上の命令は次の命令のように簡単に書くことも出来ます。

```
largestprime /@ {3, 4, 5}
```

```
{3, 2, 5}
```

実は，最初のMapの使い方も少し省略している部分があります。全てを省略しないで記述すると次のようになります。第一引数のところで，シャープ「#」やアパサンド「&」が使われています。これは純関数と呼ばれるもので，シャープ「#」は引数を，アパサンド「&」は関数定義の範囲を指定しています。

従って，次の純関数「largestprime[#] &」は，引数を関数largestprimeに渡した結果を返す関数を意味します。このような引数が一つの関数（純関数）をMapの中で利用する場合は，「[#] &」の部分を省略できることになっています。それが，Mapの最初に出てきた標準的な書き方になります。

```
Map[largestprime[#] &, {3, 4, 5}]
```
```
{3, 2, 5}
```

なお，純関数は関数Functionの省略形であり，次の様にも書けます。

```
Map[Function[largestprime[#]], {3, 4, 5}]
```
```
{3, 2, 5}
```

この他，純関数を使うと，数学的な関数を簡単に表現できます。

```
Map[#^2 + 1 &, {1, 2, 3}]
```
```
{2, 5, 10}
```

```
Map[#^2 + 2 # + 1 &, {1, 2, 3}]
```
```
{4, 9, 16}
```

もちろん， *Mathematica* にもForやWhileなどの一般的な制御構文も用意されていますが，なるべくMapを使ってください。その方が， *Mathematica* らしいし，動作も格段に速いです。なお，条件分岐を行うIfやSwitchも使えます。

演習

- リストに対して一括処理を行う関数は他にも用意されています。そのうち，次の関数について調べて，実際に使ってみましょう。

  Scan，MapThread，MapIndexed

- 関数の属性を設定することでも，リストに対して一括処理を行えます。

  次の属性Listableについてドキュメントセンターで調べてみましょう。

```
SetAttributes[largestprime, Listable]
```

```
largestprime[{3, 4, 5}]
```

```
{3, 2, 5}
```

■ 副作用（変数のスコープ）を理解しよう

次に，与えられた整数を割り切る素数のうち，大きさが最大のものと最小のものとの差を求める関数maxdiffprimeを定義してみます。

```
maxdiffprime[n_] :=
 FactorInteger[n][[-1, 1]] - FactorInteger[n][[1, 1]]
```

実際に使ってみると，きちんと7 – 2 = 5が求まっています。

```
maxdiffprime[2 * 3 * 7]
```

```
5
```

上の定義では，FactorIntegerを二回評価しています。結果は同じなので，一度評価すれば十分です。そこで，FactorIntegerを二度使わずに定義を行うことを考えます（なぜなら，その方が効率的だからです）。

先ほど習った純関数を使うと，共通しているところを引数（シャープ）を介して，次のようにひとつにまとめることが出来ます。

```
maxdiffprime[n_] :=
 Function[#[[-1, 1]] - #[[1, 1]]][FactorInteger[n]]
```

または，リストの部分取り出しを工夫して，減算を行う関数Subtractに引き渡すという方法もあります。

```
maxdiffprime[n_] :=
 Subtract @@ (FactorInteger[n][[{-1, 1}, 1]])
```

しかし，より作業内容をわかり易くするためには，次のように，与えられた整数を割り切る素数のリスト「alltheprimes」，その中で最小の素数「smallestprime」，最大の素数「largestprime」をそれぞれ定義してから，最終的に，最大と最小の差「largestprime – smallestprime」を返した方

が良いでしょう。なお，*Mathematica* ではセミコロン「;」で区切ることで複数の命令を同時に実行できることを思い出してください。

```
maxdiffprime[n_] :=
 (alltheprimes = FactorInteger[n][[All, 1]];
  smallestprime = First[alltheprimes];
  largestprime = Last[alltheprimes];
  largestprime - smallestprime)
```

このように定義した場合も，次のようにきちんと動作します。

```
maxdiffprime[2 * 3 * 7]
```

```
5
```

しかしながら，先ほど定義したはずの「与えられた整数を割り切る最大素数」を求める関数を使ってみると，何やら動作がおかしくなっています。

```
largestprime[2 * 3 * 7]
```

```
7[42]
```

そこで，いま定義したmaxdiffprimeと，先ほど定義したlargestprimeの内容を確認してみます。単語「largestprime」に注目して確認してください。

```
?maxdiffprime
```

```
Global`maxdiffprime
```

```
maxdiffprime[n_] :=
 (alltheprimes = FactorInteger[n][[All, 1]];
  smallestprime = First[alltheprimes];
  largestprime = Last[alltheprimes];
  largestprime - smallestprime)
```

```
?largestprime
```

```
Global`largestprime
```

```
Attributes[largestprime] = {Listable}
```

```
largestprime = 7
```

```
largestprime[n_] := FactorInteger[n][[-1, 1]]
```

maxdiffprimeの中で「largestprime = Last[alltheprimes]」が実行され，結果として，largestprimeという前に定義した関数の内容が上書きされているのがわかります。このように，関数の中で一時的に適当な変数を使うと，その副作用として，これまでに定義してきたものが上書きされてしまうことがあります。

実際のプログラムでは，変数の有効範囲（スコープ）を適切に定めることで，これらの副作用を最小限に抑えることが必要になってきます。*Mathematica* には，静的スコープを実現する関数Moduleが用意されており，次のように関数maxdiffprimeを定義し直すことで，副作用をなくすことが出来ます。この定義で使用されるalltheprimes，smallestprime，largestprimeは，その有効範囲がModuleの内部に限定されるため，外部にあるlargestprimeの定義を上書きすることはありません。

```
maxdiffprime[n_] :=
 Module[{alltheprimes, smallestprime, largestprime},
  alltheprimes = FactorInteger[n][[All, 1]];
  smallestprime = First[alltheprimes];
  largestprime = Last[alltheprimes];
  Return[largestprime - smallestprime]
 ]
```

簡単に関数Moduleの使い方を書くと，次のようなものになります。

Module[{一時的に利用する変数}, 実際の処理; Return[返り値]]

なお， *Mathematica* には動的スコープを実現する関数Blockも用意されていますので，必要に応じて使い分けてください。

オプションを設定しよう

ランダムに天気予報をするtenkiを作りましたが，指定した日数分の予報を行えるように改造します。例えば，次のように改造した場合を考えます。

```
tenki[n_] :=
 RandomChoice[{"晴れ", "曇り", "雨", "晴れのち曇り",
   "曇り時々雨"}, n]
```

三日間分のランダムな天気予報が欲しいとき，次のように実行します。

```
tenki[3]
```

```
{晴れ, 雨, 雨}
```

この関数をもっと良くすることを考えます。例えば，関数Styleと色の名前を組み合わせることで，文字列に簡単に色を付けることが出来ます。グラフィックスのところで扱ったオプションを自前の関数tenkiに設定して，この色分けを有効にしたり，無効にしたり出来るようにしてみましょう。

```
Style["晴れ", Blue]
```

```
晴れ
```

まず，自前の関数で利用できるオプションのリストと，各オプションのデフォルトの値を設定する必要があります。これは，関数Optionsで自分の関数を囲み，その右辺にオプションとそのデフォルト値を指定することで可能です。

ここでは，オプション名が「Coloring」でデフォルト値が「False」のオプションを，関数tenkiに設定しています。設定したいオプションが複数個ある場合は，カンマ区切りで同時に指定します。

```
Options[tenki] = {Coloring → False}
```

```
{Coloring → False}
```

関数本体には，(1) オプションを受け取れるようにする，(2) オプションの内容に応じて動作を変化出来るようにする，の二つの改造が必要になりま

す。

オプションを関数で受け取れるようにするには，アンダースコア三つ「___」を利用します。オプションは複数個同時に指定されることもあれば，何も指定されないこともあります。そのため，アンダースコアー一つ「_」のパターンでは十分ではなく，「カンマで区切られた0個以上の式」を表す「___」が必要になります。名前はなんでも構いませんが，以下の例では「opts___」と名前「opts」を付けています。

オプションの内容に応じて動作を変化させるには，指定されたオプションの値を取り出し，その値に応じて挙動を切替える必要があります。まず，指定された値の取り出しは「Coloring /. {opts} /. Options[tenki]」で行えます。オプションが変換規則として与えられているのは，このためです。切替えの方法はいろいろとありますが，以下では「If」を用いて分岐を行っています。

```
tenki[n_, opts___] := Module[{yohou},
  yohou = RandomChoice[
    {"晴れ", "曇り", "雨", "晴れのち曇り", "曇り時々雨"},
    n];
  If[(Coloring /. {opts} /. Options[tenki]) === False,
   Return[yohou]
  ];
  yohou = yohou /. {"晴れ" → Style["晴れ", Green],
     "雨" → Style["雨", Blue]};
  Return[yohou]
 ]
```

Mathematica では，等号一つ「=」で定義を，等号二つ「==」で数学的に等しいことを，等号三つ「===」で記号的に等しいことを，それぞれ表します。うまく使い分けるようにしてください。

次のようにオプションを切替えることで，色を付けたり，色を付けなかったり，と関数の動作を切替えることが出来るようになりました。

```
tenki[3]
```

{曇り, 雨, 晴れのち曇り}

```
tenki[3, Coloring → True]
```

{晴れ, 曇り時々雨, 雨}

まとめと演習

ここまでの内容を降りかえって幾つかの演習をしてみましょう。

演習

- Euclidの互除法で整数の最大公約数を求めてみる。

  Euclidの互除法は，割算を繰り返していくことで最大公約数を求めるものです。余りが0になったときの，割られる数が最大公約数になります。そこで，遅延定義を使って次のように関数euclidを定義します。

```
euclid[a_, 0] := a;
euclid[a_, b_] := euclid[b, Mod[a, b]];
```

  実際に使ってみます。

```
euclid[32, 48]
```

```
16
```

```
GCD[32, 48]
```

```
16
```

- 行列の行の基本変形をしてくれる関数を作る。

  (*i*, *j*)要素を使って，第*n*行を消去する操作を行う関数gyouを定義します。*Mathematica* の関数定義は，あくまでもパターンマッチですから，他のプログラミング言語と異なり，引数で受け取ったもの（以下の例では，mat）を直接書き換えることは出来ません。一度，違う変数（以下の

例では，mymat）に代入し，その変数（mymat）を変更していく必要があります。

```
gyou[mat_, i_, j_, n_] := Module[{mymat = mat},
  mymat[[n]] =
   mymat[[n]] - mymat[[i]] / mymat[[i, j]] *
     mymat[[n, j]];
  Return[mymat]
 ]
```

Mathematica でも，インクリメントやデクリメントを始めとする記法を使うことが出来ます。そのため，上の関数定義は，「-=」を用いて，次のようにより短く書き表すことも可能です。

```
gyou[mat_, i_, j_, n_] := Module[{mymat = mat},
  mymat[[n]] -= mymat[[i]] / mymat[[i, j]] *
    mymat[[n, j]];
  Return[mymat]
 ]
```

実際に，次の行列に対して関数を使ってみます。

```
MatrixForm[matA = {{1, 2}, {3, 4}}]
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

(1, 1)要素を使って，第2行を消去してみます。

```
MatrixForm[gyou[matA, 1, 1, 2]]
```

$$\begin{pmatrix} 1 & 2 \\ 0 & -2 \end{pmatrix}$$

- 関数を与えると「接線が動くアニメーション」を作成する関数を作る。

Mathematica では引数が指定されなかった場合に，代りに標準の値を使うことも可能です。次の関数では「stp_: Automatic」という部分で，この手法を利用しています。接線を動かす刻幅（stp）が指定されなかったと

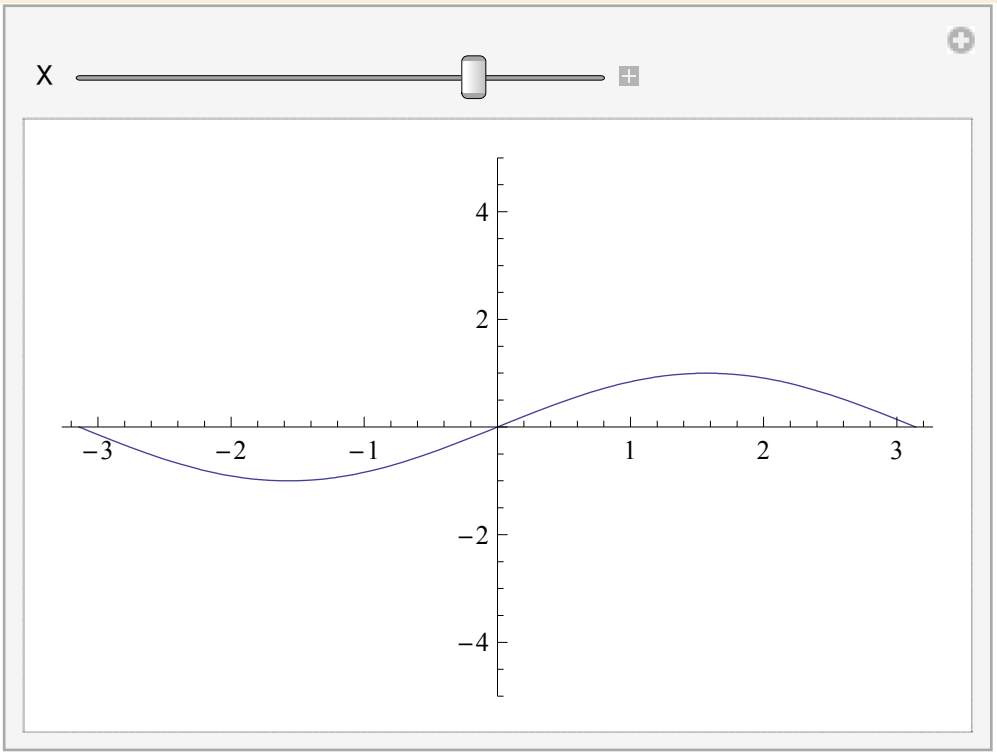
きは，代りに「stp = Automatic」という関係が成り立つようになります。これによって，関数定義の幅も広がり，以前に出てきたRangeのように引数を省略しても動作する関数を定義することが可能となります。

また，下記の関数の中で「Evaluate」という始めてみる関数が使われています。これは，その部分を強制的に（事前に）評価させる関数です。接線は，接点の*x*座標が変化する度に計算する必要があります。その過程において，傾きなどを求めるときに出てくる変数と，最終的な接線の式に出てくる変数は同じものですが，前者は事前に値が代入されていなければいけません。そのため，関数Evaluateを使って，グラフィックスを描く前に，接線の式を確定させる必要があります。

```
sessen[f_, {var_, x0_, x1_, stp_:Automatic},
  opts___] := Module[{df, xstep},
  xstep = If[NumberQ[stp], stp, (x1 - x0) / 20];
  df = D[f, var];
  Manipulate[
   Plot[
    Evaluate[
     {f, (df /. var → X) (x - X) + (f /. var → X)}],
    {x, x0, x1}, opts], {X, x0, x1, xstep}]
 ]
```

実際に動かしてみた例です。ここでは，描画範囲を固定するために，オプションPlotRangeを指定しています。上の関数ではオプションを全て関数Plotに引き継いでいるため，様々なグラフィックスに関するオプションを利用できる関数を作れています。

```
sessen[Sin[x], {x, -Pi, Pi, 0.1}, PlotRange → 5]
```



APPENDIX: 違った角度からの使い方の資料

ワープロの編集機能を真似てみよう

文章に含まれる単語の数を調べてみる

リストの話を思い出してください．文字や文字列（文字が複数連なったもの）を*Mathematica*に指示する場合，二重引用符（ダブルクオーテーション：「"」）で囲む必要がありました．同じように，食品の名前という単語だけでなく，文章を扱う場合にも二重引用符を使う必要があります．例えば，早口言葉を*Mathematica*に覚えさせる場合も次のように二重引用符を使う必要があります．

```
早口言葉`桃 = "スモモも桃も桃のうち桃もスモモももものうち"
```

```
スモモも桃も桃のうち桃もスモモももものうち
```

この文章の文字数を*Mathematica*に調べさせるには「次の文字列の文字数を調べよ」という意味を持つ「StringLength」なる命令を使います．次のように，この早口言葉の文字数は21文字であることを*Mathematica*は教えてくれます．

```
StringLength[早口言葉`桃]
```

```
21
```

この早口言葉は「も」や「桃」がたくさん出てくるので，それぞれ何回ずつ出てくるのか知りたくなりますね．そのような場合には「StringCount」という命令を使います．これは「次の文字列の中に指定した文字列が何回出てくるか調べよ」という意味の指示になります．実際に次のように*Mathematica*に問い合わせることで，「も」が6回，「桃」が3回使われていることがわかります．

```
StringCount[早口言葉`桃, "も"]
```

```
6
```

```
StringCount[早口言葉`桃, "桃"]
```

```
3
```

このStringCountは，一文字だけでなく単語や文章についても出現回数を調べる命令になっています．従って，次のように「もも」が何度出てくるかも調べることが出来ます．この早口言葉には「も」が6回も出てくるのに「もも」は1回しか出てこないことがわかります．

```
StringCount[早口言葉`桃, "もも"]
```

```
1
```

【キーボード入力の補完】*Mathematica*に指示を出す組込み関数は，英単語から構成されているので覚え易いものです．しかしながら，キーボードで「StringCount」と11文字も入力するのは面倒です．そのため*Mathematica*には「式の補完」という機能が備わっています．この機能は，WindowsとUnix(Linux)ではCTRLを押しながら「k」を押すことで，MacintoshではCMDを押しながら「k」を押すことで利用できます．例えば，StringCountを入力するには「StringCoCTRLk」でも良いですし，「StrCTRL」で表示される一覧表の中から選択することも出来ます．

文章に含まれる単語を置き換えてみる

ワープロソフトでは「置換」という機能が備わっていることは多く，文章中の特定の言葉を別の言葉に置き換えることが出来たりします．同じような*Mathematica*の機能を使って，先ほどの早口言葉の漢字をひらがなに直してみましょう．置き換えに用いる組込み関数は「StringReplace」で「次の規則に従って，文字列に含まれる単語を置き換えなさい」という意味になります．「→」は「->」（マイナス記号と大なり記号）として入力します．

```
StringReplace[早口言葉`桃, "桃" → "もも"]
```

```
スモモももももものうちももももスモモもものうち
```

【InputAutoReplacements】*Mathematica*のノートブックに入力された式は，見易くなるように自動的に変換されます．これには例えば次のようなも

のがあります.

キーボードからの入力	*Mathematica* の自動変換先
->	→
:>	:→
<=	≤
>=	≥
!=	≠
==	==

非常に読みづらい文章になってしまいましたが，加えて，カタカナもひらがなに直してみましょう．カタカナは「ス」と「モ」が使われています．これらを個別にひらがなにするには「桃」の場合と同様に次のように指示を行います．

```
StringReplace[早口言葉`桃, "ス" → "す"]
```

```
すモモも桃も桃のうち桃もすモモもももものうち
```

```
StringReplace[早口言葉`桃, "モ" → "も"]
```

```
スももも桃も桃のうち桃もスももももものうち
```

．．．意味がないですね．「桃，ス，モ」の全てを同時にひらがなにしたいですね．このような場合，置き換えの規則（"桃"→"もも", "ス"→"す", "モ"→"も"）をリストで一括指定します．このように，*Mathematica*への指示では単一のものを複数組み合わせるときにもリストが活躍します．

```
StringReplace[早口言葉`桃,
 {"桃" → "もも", "ス" → "す", "モ" → "も"}]
```

```
すももももももももものうちももももすももももものうち
```

この状態で先ほど使った組込み関数で文字の数を調べると，音としての「も」の回数がわかります．*Mathematica*が調べた結果を次の指示で使うためには，「*Mathematica*が直前に報告した結果」を意味する「%」を使うと非常に便利です．もちろん，「%」の代わりに直接「StringReplace[早口言葉`桃,{"桃"→"もも", "ス"→"す", "モ"→"も"}]」を使うことも出来ます．

```
StringCount[% , "も"]
```

```
16
```

```
StringLength[早口言葉`桃]
```

```
21
```

このようにして調べると，この早口言葉は21文字中16文字も同じ「も」が出てくることがわかります．面白いですね．

【前の結果の引用】*Mathematica*を使うことは「*Mathematica*との会話」です．人間同士の会話から代名詞がなくなったら非常に不便なのと同様に，*Mathematica*を使う上でも代名詞は非常に重要です．今回登場した「%」は「直前の結果」を表すものですが，これ以外にも「%%」で「二つ前の結果」，「%%%」で「三つ前の結果」を表します．即ち，パーセント記号を並べただけ前の結果を遡って引用することが出来ます．

文章に含まれる似通った単語の数を調べてみる

ワープロの置換機能に似た方法で音として「も」がたくさん入っていることがわかりました．しかし，私たちがいま使っているのはハイテクの塊とも言える*Mathematica*なのに，少しローテク（地味で時代後れ）な方法のように思えませんか．実は，もっとスマートな方法があります．

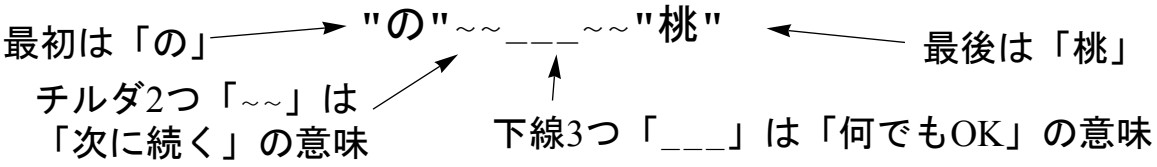
```
早口言葉`桃 = "スモモも桃も桃のうち桃もスモモももものうち"
```

いま取り上げている早口言葉の定義

*Mathematica*の組込み関数「StringCases」は「次の文字列の中から，指定したパターンにマッチする単語（部分文字列）を取り出せ」という意味の指示になります．パターンとは例えば，「の」で始まって「桃」で終わる文字列，のような条件のことを指します．この場合，*Mathematica*への命令は次のようになります．

```
StringCases[早口言葉`桃, "の" ~~ ___ ~~ "桃"]
```

```
{のうち桃}
```



文字列パターン：「の」で始まって「桃」で終わる文字列

他のパターンでも調べてみましょう．次の命令では，「桃」または「もも」または「モモ」にマッチする文字列，を調べさせています．この早口言葉には音として「もも」を6個も含んでいることがわかります．縦線（「|」）を「または」と読むことで，文字列パターンは日本語での意味と同じになることがわかると思います．

```
StringCases[早口言葉`桃, "桃" | "もも" | "モモ"]
```

```
{モモ, 桃, 桃, 桃, モモ, もも}
```

*Mathematica*の調査結果と早口言葉を比較することで，「ももも」の部分が1つ分の「もも」としか数えられていないことがわかると思います．*Mathematica*に重複している部分も数えるように補足指示（オプション）を出すことも出来ます．命令の最後に「Overlaps→True」という補足指示を加えることで，次のように重複部分も調べた結果を教えてくれます．

```
StringCases[早口言葉`桃, "桃" | "もも" | "モモ",
 Overlaps → True]
```

```
{モモ, 桃, 桃, 桃, モモ, もも, もも}
```

音として「もも」になるようなカタカナとひらがなの組合せも重複して調べさせることも可能です．この場合，次のように文字列パターンは長くなってしまいます．

```
StringCases[早口言葉`桃,
 "桃" | "もも" | "モモ" | "もモ" | "モも", Overlaps → True]
```

```
{モモ, モも, 桃, 桃, 桃, モモ, モも, もも, もも}
```

カタカナとひらがなの組合せの「もも」を効率良く*Mathematica*に指示するには，または（「|」）と次に続く「（~~）」を組み合わせて指示を出しま

す．丸括弧（「(」と「)」）は「または」の対象を明確に示すために必要です．この丸括弧がないと，1文字目が「桃」か「も」か「モ」で，2文字目が「も」か「モ」の単語，という意味になってしまいます．*Mathematica*での丸括弧は，このように物事の優先順位を定めるために使われます．

```
StringCases[早口言葉`桃, "桃" | ("も" | "モ" ~~ "も" | "モ"),
 Overlaps → True]
```

```
{モモ, モも, 桃, 桃, 桃, モモ, モも, もも, もも}
```

より正確に「もも」という音が何回出てくるかを調べるには，次のように複雑な文字列パターンを使わなければいけませんが，このパターンの意味は自分で考えてみてください．「リストの要素数を調べよ」という意味の命令「Length」を使って，13個の「もも」という音が隠れていることがわかりました．21文字中に13個もの「もも」です．これでは言いづらいのにも納得ですね．

```
StringCases[早口言葉`桃,
 "桃" | ({"も", "モ"} ~~ {"も", "モ"}) |
  ({"も", "モ"} ~~ {"桃"}) | ({"桃"} ~~ {"も", "モ"}),
 Overlaps → All]
```

```
{モモ, モも, も桃, 桃, 桃も, も桃,
 桃, 桃, 桃も, モモ, モも, もも, もも}
```

```
Length[%]
```

```
13
```

なお，オプションの「Overlaps→All」は「桃」と「桃も」のようにマッチする文字列が他のマッチする文字列に含まれる場合も別のものとして扱うという追加指示になります．これがないと13個ではなく11個しか検出されません．

【文字列パターン】*Mathematica*では，様々な種類のパターンを利用することが出来ます．早口言葉の件で扱っている文字列パターンはそのひとつです．文字列パターンについての詳しい情報はドキュメントセンターで

「StringExpression」を調べると書いてあります．なお，文字列パターンは「StringCount」とも組み合わせることが出来ます．「StringCases」の代わりに「StringCount」を使うと，同時に「Length」も使ったのと同じように，パターンにマッチする単語の個数を調べてくれます．ぜひ試してみてください．

お猿さんはシェークスピアの話を書けるか？

言葉（ひらなが，カタカナ，記号）のグループを作ろう

*Mathematica*で言葉のリストを作るのには「CharacterRange」という命令を使います．この命令の意味は「次の2つの文字の間にある全ての文字を調べよ」になります．例えば，「な」と「の」の間にある全ての文字を調べさせるには次のように命令します．結果はリストで報告されますが，当然「なにぬねの」です．

```
CharacterRange["な", "の"]
```

```
{な, に, ぬ, ね, の}
```

この機能を使うと，ひらがなのリストを作ることが出来ます．「ひらがな」という名前でひらがなのリストを覚えさせておきましょう．なお，下記の「ぁ」は小さい「あ」であって，大きな「あ」ではないので注意してください．

```
ひらがな = CharacterRange["ぁ", "ん"]
```

```
{ぁ, あ, ぃ, い, ぅ, う, ぇ, え, ぉ, お, か, が, き,
 ぎ, く, ぐ, け, げ, こ, ご, さ, ざ, し, じ, す, ず, せ,
 ぜ, そ, ぞ, た, だ, ち, ぢ, っ, つ, づ, て, で, と, ど,
 な, に, ぬ, ね, の, は, ば, ぱ, ひ, び, ぴ, ふ, ぶ, ぷ,
 へ, べ, ぺ, ほ, ぼ, ぽ, ま, み, む, め, も, ゃ, や, ゅ,
 ゆ, ょ, よ, ら, り, る, れ, ろ, ゎ, わ, ゐ, ゑ, を, ん}
```

同様に，カタカナのリストも「カタカナ」という名前で覚えさせておきましょう．「ヴ」が最後に来ていますが，これは「ン」でも良いと思います．なお，ひらがなのときと同じで，下記の「ァ」は小さい「ア」であって，大きな「ア」ではないので注意してください．

```
カタカナ = CharacterRange["ァ", "ヴ"]
```

```
{ァ, ア, ィ, イ, ゥ, ウ, ェ, エ, ォ, オ, カ, ガ, キ, ギ,
 ク, グ, ケ, ゲ, コ, ゴ, サ, ザ, シ, ジ, ス, ズ, セ, ゼ,
 ソ, ゾ, タ, ダ, チ, ヂ, ッ, ツ, ヅ, テ, デ, ト, ド, ナ,
 ニ, ヌ, ネ, ノ, ハ, バ, パ, ヒ, ビ, ピ, フ, ブ, プ, ヘ,
 ベ, ペ, ホ, ボ, ポ, マ, ミ, ム, メ, モ, ャ, ヤ, ュ, ユ,
 ョ, ヨ, ラ, リ, ル, レ, ロ, ヮ, ワ, ヰ, ヱ, ヲ, ン, ヴ}
```

記号については，順序が複雑で一括して作り出すことが難しいので，リスト（グループ）として必要な記号を「記号」という名前で覚えさせておきます．

```
記号 = {", ", ". ", "「", "」", "（", "）"}
```

```
{, , . , 「, 」, （, ）}
```

ちなみに半角のアルファベットも次のように生成することが出来ます．

```
CharacterRange["a", "z"]
```

```
{a, b, c, d, e, f, g, h, i, j, k, l,
 m, n, o, p, q, r, s, t, u, v, w, x, y, z}
```

```
CharacterRange["A", "Z"]
```

```
{A, B, C, D, E, F, G, H, I, J, K, L,
 M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z}
```

言葉のグループからランダムな単語を作ってみよう

この章では，お猿さんがキーボードを使ってシェークスピアの文章を入力する可能性は皆無ではない，という有名な話を実際に確かめてみようとしています．言い替えれば，ランダムに選ばれた語から構成される文章が意味を持つことがあるか，ということですね．もっとも，我々人間に確認可能な短い時間で，お猿さんがシェークスピアを入力することは非常に難しいと思いますが．．．

この実験にちょうど良い命令として「RandomChoice」があります．意味は「次のグループの中から無作為に要素を取り出し，取り出した要素を元に戻す．これを指示された回数実験せよ」です．ひらがなのリストからランダム（無作為）に3個のひらがなを取り出すには，次のように命令することになります．3個のひらがなはランダムに選ばれるため，*Mathematica*に命令を出すたび（評価を行うごと）に結果は変化します．

```
RandomChoice[ひらがな, 3]
```

```
{く, え, せ}
```

このままでは「語のリスト」であって「単語」になっていません．そこで，*Mathematica*に対して「次の語のリストを単語にまとめなさい」という意味の「StringJoin」なる命令も併せて実行させることにします．次の例のように，ひらがな5つから構成される単語が評価毎に作られていきます．

```
StringJoin[RandomChoice[ひらがな, 5]]
```

```
うふぺゃり
```

同様に，カタカナの単語，記号のみからなる文字列なども作り出すことが出来ます．

```
StringJoin[RandomChoice[カタカナ, 5]]
```

```
ガオタザェ
```

```
StringJoin[RandomChoice[記号, 5]]
```

```
) (, . )
```

ちなみに，似たような命令に「RandomSample」があります．この意味は「次のグループの中から無作為に要素を取り出す（元には戻さない）．これを指示された回数実験せよ」です．従って，結果は非常に良く似ていますが，いくら繰り返しても「きつつき」，「とまと」，「とうきょう」などは出てきません．

```
StringJoin[RandomSample[ひらがな, 3]]
```

ませで

【ランダムな選択】*Mathematica*には最初のバージョンから疑似乱数を作り出す命令「Random」が組み込まれていました．最新の*Mathematica* 6では，より使い易く目的に応じた6つの命令「RandomChoice, RandomComplex, RandomInteger, RandomPrime, RandomReal, RandomSample」に進化しています．今回使用したものは，以前のバージョンになかった組込み関数になります．

言葉のグループからランダムな文章を作ってみよう

漢字はありませんが，ひらがな，カタカナ，記号だけでも文章を作るには十分です．例えば，冒頭で出てきた早口言葉で漢字を使わない場合，次のようになりますが十分理解できます（分かり易いかは別問題とすれば）．

```
StringReplace[早口言葉`桃, "桃" → "もも"]
```

スモモももももものうちももももスモモもものうち

以前に使った組込み関数Unionで，文章の構成要素となるリスト「言葉のもと」を作っておけば，単語を作るようにRandomChoiceとStringJoinにより任意の長さの文章を作り出すことが出来ます．

```
言葉のもと = Union[ひらがな, カタカナ, 記号];
```

```
StringJoin[RandomChoice[言葉のもと, 30]]
```

ブカンとヂろけゆな「ざヤノぬをゃぉせケゾルをスヌびワなシテミ

しかし，これでは文章には程遠いですね．そこで，文章はひらがなの塊とカタカナの塊と記号から構成されていると仮定しましょう．塊の長さ（1文字から5文字程度）もランダムに決める必要があります．文字数は整数（…, −2, −1, 0, 1, 2, …）なので，単語作りに使った組込み関数に似た命令の「RandomInteger」により「次の範囲から整数を無作為に取り出せ」という指示を*Mathematica*に出すことが出来ます．次の例は，−5から5の整数（−5, −4, −3, −2, −1, 0, 1, 2, 3, 4, 5）から無作為にひとつ選ばれます．

```
RandomInteger[{-5, 5}]
```

```
1
```

この組込み関数を使って「塊の長さ」を覚えさせておきましょう．まず，長さとしては1文字から5文字とし，RandomIntegerへの指示は{1,5}となります．塊の長さは評価するたびに変化する可能性（無作為に整数が選ばれるため）があるので，ショートカットやリンクに準ずる遅延的な定義（コロンとイコール）を使って定義します．

```
塊の長さ := RandomInteger[{1, 5}]
```

```
塊の長さ
```

```
5
```

```
塊の長さ
```

```
5
```

定義した塊の長さを使って，ひらがなの塊も定義しましょう．基本的にはひらがなの単語を作ったときと同じですが，無作為に取り出す回数を「塊の長さ」とすることと，遅延的な定義（コロンとイコール）を使うこと点で異なります．同じようにカタカナの塊も定義しておきますが，記号については常に1文字になるように定義しておきます．実際に，ひらながの塊やカタカナの塊を評価すると，無作為な言葉の塊が出てくることを確認できます．

```
ひらがなの塊 :=
 StringJoin[RandomChoice[ひらがな, 塊の長さ]]
```

```
カタカナの塊 :=
 StringJoin[RandomChoice[カタカナ, 塊の長さ]]
```

```
記号の欠片 := StringJoin[RandomChoice[記号, 1]]
```

ひらがなの塊

らあよ

カタカナの塊

ナ

さて，これらの塊を使って文章を作るのですが，ひらがな，カタカナ，記号をどのような割合いでどのような順番で組み合わせるかという問題が残っています．特に，普通の文章に含まれる記号の割合いは著しく低いですし，カタカナもひらがなよりは低いでしょう．これを反映させなければいけません．何度も使っている*Mathematica*の組込み関数「RandomChoice」では，選ばれる確率を指定することも出来ます．例えば，次の例では「言葉のもとからひとつ」を遅延的に定義していますが，ひらがなの塊が70%程度，カタカナの塊が20%程度，記号の欠片が10%程度の割合いで選択されるように指示しています．

```
言葉のもとからひとつ :=
 RandomChoice[
  {0.7, 0.2, 0.1} → {ひらがなの塊, カタカナの塊, 記号の欠片}]
```

言葉のもとからひとつ

れず

文章にするには，この「言葉のもとからひとつ」をいくつも組み合わせることになります．そのためには「次の指示を指定された回数だけ繰り返せ」という意味の「Table」を使うことになります．この組込み関数は非常に便利なので，今後何度も使うことになります．例えば，言葉のもとからひとつを10回繰り返したい場合，次のように「{10}」という指示を出します．この数字を変えれば繰り返す回数を変化させることが可能です．

```
Table[言葉のもとからひとつ, {10}]
```

{ヤ，ナタクポ，ぉぴさ，ポュモ，ぅ，
 しが，ねぉ，ぁしぇ，ゅよぉづ，どやわゎで}

文章にするには，全体をStringJoinで囲めば良いので，最終的に次のような命令でランダムな文章を作ることが出来るようになります．何度も評価を繰り返すと，普段の会話のような文章が表れるかも，しれません．．．

```
StringJoin[Table[言葉のもとからひとつ, {10}]]
```

ゆだげてだェギィヤムボトメびわずぇけえシゲヴチぉぎだジへガセず

【遅延的な定義の本質】*Mathematica*での「即時的な定義」と「遅延的な定義」の違いは，ファイルマネージャにおける「コピー」と「ショートカット」の違いに近い，という話をしました．より正確に表現すると，イコール（=）だけを使った即時的な定義では，右辺の評価結果を覚えますが，コロンとイコール（:=）を使った遅延的な定義では，右辺の評価はせずに右辺をそのまま覚えます．従って，無作為に何を選択する組込み関数と即時的な定義を組み合わせても，毎回同じ結果しか表示されませんが，遅延的な定義を組み合わせれば，毎回違うランダムな結果が表示されることになります．

小学校で習う漢字も追加しよう

やはり漢字がないと寂しいので，小学校の最初に習う漢字を追加しましょう．文部科学省のウェブサイトでは，各種の指導要領が公開されています．その中から小学校学習指導要領の国語に関するところに掲載されている「学年別漢字配当表」から，第一学年のものを写したのが次の文字列です．

```
小学校学習指導要領一学年漢字 =
  "一右雨円王音下火花貝学気九休玉金空月犬見五口校左三山子四糸
    字耳七車手十出女小上森人水正生青夕石赤千川先早草足村大
    男竹中虫町天田土二日入年白八百文木本名目立力林六";
```

ひらがなやカタカナなどは一語ずつのリストになっていましたので，この漢字についても一語ずつのリストに直す必要があります．手作業で行うことも可能ですが，*Mathematica*の「Characters」という命令を使うことで，次のように簡単に1文字ずつに分割できます．この命令は「次の文字列を一語ずつの文字のリストに分解せよ」という意味なのは明らかでしょう．

```
小学校一学年 = Characters[小学校学習指導要領一学年漢字]
```

```
{一, 右, 雨, 円, 王, 音, 下, 火, 花, 貝, 学, 気, 九,
 休, 玉, 金, 空, 月, 犬, 見, 五, 口, 校, 左, 三, 山,
 子, 四, 糸, 字, 耳, 七, 車, 手, 十, 出, 女, 小, 上,
 森, 人, 水, 正, 生, 青, 夕, 石, 赤, 千, 川, 先, 早,
 草, 足, 村, 大, 男, 竹, 中, 虫, 町, 天, 田, 土, 二, 日,
 入, 年, 白, 八, 百, 文, 木, 本, 名, 目, 立, 力, 林, 六}
```

あとは先ほどと同様に「小学校一学年の塊」を遅延的な方法で定義しておきます.

```
小学校一学年の塊 :=
 StringJoin[RandomChoice[小学校一学年, 塊の長さ]]
```

「言葉のもとからひとつ」の定義にも漢字を入れて定義し直しましょう. 割合いは, ひらがなが40%, 漢字が30%, カタカナが20%, 記号が10%にしてみました. 結果を見ると, 多少は文章らしくなってきたのではないでしょうか.

```
言葉のもとからひとつ :=
 RandomChoice[{0.4, 0.3, 0.2, 0.1} →
   {ひらがなの塊, 小学校一学年の塊, カタカナの塊, 記号の欠片}]
```

```
StringJoin[Table[言葉のもとからひとつ, {10}]]
```

```
いぇモバ千立出うしが林森下王目ズブアグキせこア
```

文章として意味のあるものになることは非常に稀ですが, 漢字の塊だけを次のように20組程度作ると, 中には意味のあるものが作られたり, 意味がありそうな不思議な熟語になっていたりします. もしかすると, 本当に猿もシェークスピアが書けてしまうかもしれませんね.

```
Table[小学校一学年の塊, {20}]
```

```
{天円町, 名力大森立, 青, 山九赤, 犬休白夕先, 本口六右早,
 早車町九, 木男気下, 林, 木田火口空, 手生石三, 上手本四,
 七, 校, 休中, 耳車玉女, 空六, 左見年車正, 土川, 七青}
```

ランダム作文で洒落た短文を書いてもらおう

いつ，誰が，どうした？

キーボードを無作為に押していくことでシェークスピアを書くのは無謀だということがわかりました．ランダムな熟語を作るという試みとしては面白かったと思いますが，現実的な作文にはなっていませんでした．そこで，よりシンプルに短い文を作ることにしましょう．「いつ，誰が，どうした」というシンプルな作文です．まずは，それぞれの候補をリストで定義しておきます．

```
いつ = {"昨日の朝，", "昨日の昼間，", "昨日の夕方，",
    "昨日の夜，", "今日の朝，", "今しがた，"};
```

```
誰が = {"私は", "あなたは", "先輩は", "後輩は", "知合いは"};
```

```
どうした = {"寝坊した．", "遅刻した．", "宝くじに当たった．",
    "宝くじに外れた．", "ベッドから落ちた．",
    "ごはんを食べた．"};
```

それぞれの候補の中から無作為に選択する命令は既に使った`RandomChoice`で行えます．前回使用したときとは異なり，取り出す要素の個数は1個なので，わざわざ「`RandomChoice[いつ,1]`」と*Mathematica*に指示を出す必要はありません．

```
RandomChoice[いつ]
```

```
昨日の朝，
```

文章を構成するには，残りも併せて3つの要素をそれぞれ無作為に選択すれば良いことがわかります．これを遅延的な定義で次のように覚えさせておきましょう．扱う部品が3つあるのでリストでまとめてあることに注意して

ください．遅延的な定義なので，評価するたびにランダムな文章が作られるようになります．

```
いつ誰がどうしたリスト :=
 {RandomChoice[いつ], RandomChoice[誰が],
  RandomChoice[どうした]}
```

```
いつ誰がどうしたリスト
```

```
{昨日の夜，，先輩は，ベッドから落ちた．}
```

```
いつ誰がどうしたリスト
```

```
{昨日の夜，，あなたは，寝坊した．}
```

このままでは1つの文章（文字列）になっていないので，前回も使用したStringJoinを使って結合することにしましょう．これも遅延的な定義を使って覚えさせておくことで，評価するたびにランダムに文章が作られるようになります．

```
いつ誰がどうした := StringJoin[いつ誰がどうしたリスト]
```

```
いつ誰がどうした
```

```
昨日の夕方，あなたは宝くじに当たった．
```

```
いつ誰がどうした
```

```
昨日の昼間，あなたは宝くじに外れた．
```

*Mathematica*には様々な省略表記があり，StringJoinという指示は「<>」（小なり記号，大なり記号）を使っても同じ意味になります．従って，先ほど定義した「いつ誰がどうした」という名前のショートカット（後述しますが，本来は変数ないしは関数と呼ぶ）は次のように定義することも可能です．

```
いつ誰がどうした := RandomChoice[いつ] <> RandomChoice[誰が] <>
  RandomChoice[どうした]
```

```
いつ誰がどうした
```

昨日の夜，後輩は宝くじに外れた．

【マルチパラダイム】このように，*Mathematica*では同じ意味の指示を出すのに複数の具体的な命令が存在します．書き方によって，*Mathematica*が指示を実行するために必要な時間が大きく変化するため，科学技術計算などの速度が求められる場合には注意深く記述する必要があります．一方，これまでに取り組んできた内容のように高速に計算される必要がない場合は，ユーザが理解しやすい方法で柔軟に書くことが出来るという大きなメリットになります．

いつ，誰が，何を，どうして，どうなった？

いつ，誰が，どうした，では短すぎて面白みが少ないかもしれません．もっと長い文章が生成されるように，いつ，誰が，何を，どうして，どうなった，という構成要素に拡張しましょう．まずは，先ほどと同じように各グループ毎に名前をつけて定義しておきます．各グループの要素が変化することはありませんので，即時的なもの（コピー）で大丈夫です．

```
いつ = {"昨日の朝，", "昨日の昼間，", "昨日の夕方，",
    "昨日の夜，", "今日の朝，", "今しがた，"};
```

```
誰が = {"私は", "あなたは", "先輩は", "後輩は", "知合いは"};
```

```
何を = {"パソコンを", "財布を", "ケーキを", "コーヒーを",
    "歯磨き粉を", "かばんを"};
```

```
どうして = {"壊して", "失くして", "食べて", "こぼして",
    "搾って", "背負って"};
```

```
どうなった = {"非常に困った．", "交番に届け出た．",
    "幸せだった．", "火傷した．", "歯磨きをした．",
    "買い物に行った．"};
```

簡単のため，冗長な方法で「いつ誰が何をどうしてどうなった」を遅延的に定義しましょう．前回と同じくRandomChoiceとStringJoinを使っています．繰り返しになりますが，毎回，異なる言葉が無作為に選択されないといけませんから，遅延的な定義にする必要があることに注意してください．

```
いつ誰が何をどうしてどうなったリスト :=
 {RandomChoice[いつ], RandomChoice[誰が],
  RandomChoice[何を], RandomChoice[どうして],
  RandomChoice[どうなった]}
```

```
いつ誰が何をどうしてどうなった :=
 StringJoin[いつ誰が何をどうしてどうなったリスト]
```

実際に「いつ誰が何をどうしてどうなった」を評価すると，次のように面白い文章が毎回ランダムに生成されることがわかります．

```
いつ誰が何をどうしてどうなった
```

昨日の夕方，知合いはパソコンをこぼして火傷した．

```
いつ誰が何をどうしてどうなった
```

昨日の昼間，あなたはパソコンを壊して非常に困った．

言葉を選んでくるグループが5個なのでRandomChoiceを5回記述して，それをStringJoinでまとめるという定義を行っていますが，これが10個になれば10回記述する必要が出てきます．*Mathematica*は計算機なので面倒とは思わないかもしれませんが，私たちは人間ですから同じことを何度もするのは面倒です．そのため，このような同じ操作を繰り返すことを簡単に指示できる仕組みが準備されています．まずは「いつ誰が何をどうしてどうなったリスト」の定義を確認して，共通部分がどこにあるのか，何を冗長に何度も繰り返しているかを確認しましょう．

```
? いつ誰が何をどうしてどうなったリスト
```

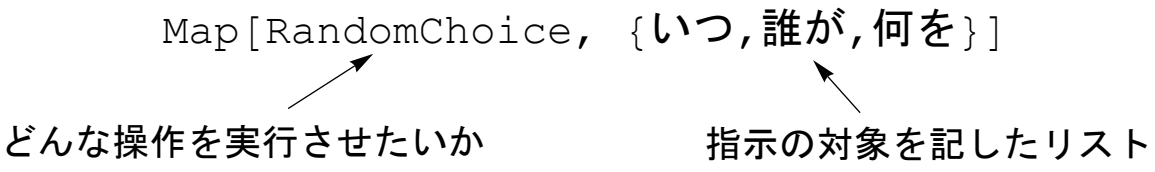
```
Global`いつ誰が何をどうしてどうなったリスト
```

```
いつ誰が何をどうしてどうなったリスト := {RandomChoice[いつ],
  RandomChoice[誰が], RandomChoice[何を],
  RandomChoice[どうして], RandomChoice[どうなった]}
```

赤字で強調表示している部分が共通している部分です．いつ，誰が，何を，どうして，どうなった，のそれぞれに対してRandomChoiceの指示を出しています．これと同じ指示を組込み関数「Map」を使うと簡単に書くことが出来ます．この命令の意味は「次のリストのそれぞれに対して同じ命令を実行せよ」です．この関数を使うと「いつ誰が何をどうしてどうなったリスト」は次のように書き直すことが出来ます．

```
Map[RandomChoice, {いつ, 誰が, 何を, どうして, どうなった}]
```

```
{昨日の夕方, , 私は, ケーキを, こぼして, 火傷した. }
```



リストの一括処理：いくつものリストに同じ操作を実行

この命令を使って「いつ誰が何をどうしてどうなった」を再度定義しなおすと次のように短い表現になります．わかりやすくなったはずですが，新しい命令であるMapを理解できていないと難しく感じるかもしれません．この命令は非常に重要で，今後も何度も出てきます．何度も読み返して便利さを確認してください．

```
いつ誰が何をどうしてどうなった :=
  StringJoin[Map[RandomChoice,
    {いつ, 誰が, 何を, どうして, どうなった}]]
```

いつ誰が何をどうしてどうなった

今しがた，知合いは財布を失くして幸せだった．

【Mapによるリスト処理】*Mathematica*はLispの流れを汲む言語と考えるのが自然で，リストで表現されたデータに対する一括処理に秀でています．ここでは単純に命令を短くする目的だけにMapを使用していますが，後の章では計算速度を速くするためにも用います．前の方の章で何度も断っていますが，*Mathematica*を使う上でリストは非常に重要な概念ですので，この一括操作も非常に重要な概念になります．始めにリストありき，という考え方を身につけることは*Mathematica*を覚える近道です．

「いつ誰がどうした？」で時制もランダムにしたい

前項まで取り上げた作文では，時制や助詞も含めてある言葉から無作為に選択していました．これでは，時制や助詞の分だけ全てを想定した言葉を事前にリストで定義しておかなければいけません．もっと手軽に時制を扱える方法がないかを考えてみたいと思います．まずは，時制に関係のない部分だけを*Mathematica*に覚えておいてもらいましょう．

```
誰が = {"私", "彼", "彼女", "友達"};
```

```
助詞 = {"が", "は", "も"};
```

```
どうした = {"勉強", "食事", "カラオケ", "寝坊"};
```

次に，時制の部分が除かれた無作為な文章を生成する「誰がどうした」を定義しましょう．毎回異なる文章を生成させるには，遅延的な定義になることに注意してください．

```
誰がどうした :=
 StringJoin[Map[RandomChoice, {誰が, 助詞, どうした}]]
```

実際に「誰がどうした」を*Mathematica*に問い合わせることで，時制と述語が不足している文章がランダムに返ってきます．あとは，この文章にランダムに時制と述語を付けられれば何も問題ありませんが，時制と述語は「昨日，した」と「明日，する」とペアになるので，各々を勝手に述語を決める訳にもいきません．どうすれば良いでしょうか．

誰がどうした

彼は食事

誰がどうした

私が寝坊

まずは，その出来事が昨日のことであるかを判定する命令「昨日のこと？」を作りましょう（はてなマークは全角にしてください）．*Mathematica*では「`True`」という言葉で「その通り！」を意味する「真」を，「`False`」で「そうじゃない！」を意味する「偽」を表します．従って，真偽を無作為に選択する「昨日のこと？」は次のような遅延的な定義になります．何度も出てきている`RandomChoice`を使っていることに注意してください．

```
昨日のこと？ := RandomChoice[{True, False}]
```

実際に*Mathematica*へ問い合わせてみると，評価するたびに真と偽が無作為に報告されるのがわかると思います．

昨日のこと？

```
False
```

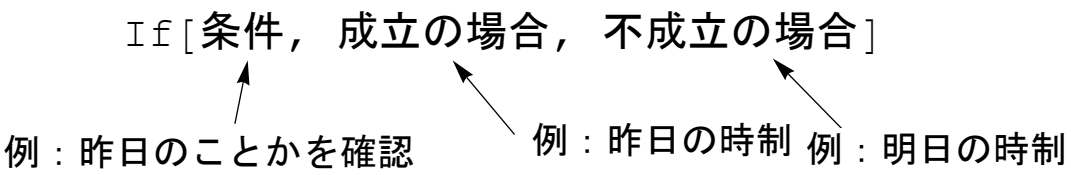
昨日のこと？

```
True
```

不足している時制と述語のペアを文章に追加するには，「昨日のこと？」が真であるか偽であるかという状況に応じて*Mathematica*がすべきことを変化させる必要があります．これには，「次の条件が満たされたら，指定の命令を実行せよ」という意味の「`If`」を使います．昨日のことであれば「昨日，」と「した．」を追加し，そうでなければ「明日，」と「する．」を追加しています．

```
If[昨日のこと？, StringJoin["昨日，", 誰がどうした,
  "した．"], StringJoin["明日，", 誰がどうした, "する．"]]
```

明日，彼は寝坊する．



条件分岐：内容によって命令を変える

最終的に，時制も無作為に選択される「いつ誰がどうした」は次のように遅延的な定義で表現されることになります．実際に評価してみると，いくつか面白い文章が作られることがわかると思います．

```
いつ誰がどうした :=
 If[昨日のこと？, StringJoin["昨日，", 誰がどうした,
   "した．"], StringJoin["明日，", 誰がどうした, "する．"]]
```

```
いつ誰がどうした
```

明日，彼が寝坊する．

```
いつ誰がどうした
```

昨日，友達が寝坊した．

【TrueとFalse】*Mathematica*では条件が満たされていることを「True」で表し，条件が満たされていないことを「False」で表します．いわゆる真偽値と呼ばれるもので，今回のような条件分岐の他，以前にも出てきたような判定問題（リストに指定した要素が含まれているか）などでの*Mathematica*からの報告にも使われます．今後も頻繁に出てきますので，良く覚えておいてください．

「いつ誰が何をどうしてどうなった？」で時制もランダムにしたい

前項よりもバリエーションを増やして文章も長くすることを考えましょう．まずは，前項と同じように時制を除いた「誰が何をどうしてどうなった」をこれまでと同じように定義しましょう．

```
誰が = {"私", "彼", "彼女", "友達"};
```

```
助詞 = {"が", "は", "も"};
```

```
何を = {"数学を", "運動会を", "十八番を", "携帯電話を",
    "卵を"};
```

```
どうして = {"勉強して", "食べて", "歌って", "寝過ごして",
    "壊して", "落として"};
```

```
何が = {"成績を", "調子を", "懐を", "雰囲気を", "気分を"};
```

```
どうなった = {"良く", "悪く", "寂しく", "悲しく"};
```

```
誰が何をどうしてどうなった :=
 StringJoin[Map[RandomChoice,
   {誰が, 助詞, 何を, どうして, 何が, どうなった}]]
```

この時点で「誰が何をどうしてどうなった」を*Mathematica*に評価させると，これまでと同じように，時制の部分を除いた文章がランダムに返ってきます．

```
誰が何をどうしてどうなった
```

```
私も運動会を歌って気分を悲しく
```

時制と述語を加えて，私達としては「明日，（本文）する．」とか「昨日，（本文）した．」という文章にしたいですね．そこで，このような時制と述語のペアをリストで定義しておきましょう．リストには「時制」という名前を付けて*Mathematica*に覚えておいてもらいます．

```
時制 = {"昔，（本文）したものだ．", "昨日，（本文）した．",
    "今日，（本文）したところだ．", "明日，（本文）する．",
    "将来，（本文）したい．"};
```

いつの文章であるかをランダムにしたいので，時制をランダムに選択するように「時制の選択」も定義しておきます．そうすると，次のように時制と述語のペアが無作為に返ってくることがわかると思います．遅延的な定義を使うのにも慣れてきたでしょうか．

```
時制の選択 := RandomChoice[時制]
```

```
時制の選択
```

明日，（本文）する．

```
時制の選択
```

将来，（本文）したい．

「（本文）」の部分が例えば「私は数学を勉強して成績を良く」になれば，作りたかった文章に近づきます．そこで，時制がランダムに選択されるようにしたままで，（本文）を置き換えるよう*Mathematica*に指示を出してみます．次のように，以前に利用した命令`StringReplace`を使うことで，簡単に置き換えは行えます．

```
StringReplace[時制の選択,
 "（本文）" → "私は数学を勉強して成績を良く"]
```

明日，私は数学を勉強して成績を良くする．

このままでは，本文のところがランダムに選択されないので，置き換え先をランダムに本文を生成する「誰が何をどうしてどうなった」に置き換えてみます．すると，次のように時制と本文が無作為に選択された文章が作られるようになります．

```
StringReplace[時制の選択,
 "（本文）" → 誰が何をどうしてどうなった]
```

```
将来，友達は数学を食べて雰囲気を良くしたい.
```

あとは遅延的な定義で「いつ誰が何をどうしてどうなった」を次のように定義することで，時制も本文も無作為に選択された文章をつくり出すことが出来ます.

```
いつ誰が何をどうしてどうなった :=
 StringReplace[時制の選択,
  "（本文）" → 誰が何をどうしてどうなった]
```

```
いつ誰が何をどうしてどうなった
```

```
昨日，友達は十八番を寝過ごして気分を悪しくした.
```

■ 文豪の作品を小学生になって読んでみよう

▫ 作品に使われている文字を調べてみよう

今回の目的は文豪の作品を調べることです．例題として夏目漱石の「吾輩は猫である」から最初の段落を取り上げたいと思います．自分で試してみる際には，青空文庫（http://www.aozora.gr.jp）などからお気に入りの作品を探してみることをお薦めします．まずは，文章を分かり易い名前で*Mathematica*に覚えさせましょう．このような文章の定義の際などは，最後にセミコロン（;）を付けておくと復唱（結果出力）がないのでスッキリするので，忘れずに付けておきましょう.

```
吾輩は猫である = 
  "吾輩は猫である。名前はまだ無い。どこで生れたかとんと見当がつかぬ。何でも薄暗いじめじめした所でニャーニャー泣いていた事だけは記憶している。吾輩はここで始めて人間というものを見た。しかもあとで聞くとそれは書生という人間中で一番獰悪な種族であったそうだ。この書生というのは時々我々を捕えて煮て食うという話である。しかしその当時は何という考もなかったから別段恐しいとも思わなかった。ただ彼の掌に載せられてスーと持ち上げられた時何だかフワフワした感じがあったばかりである。掌の上で少し落ちついて書生の顔を見たのがいわゆる人間というものの見始であろう。この時妙なものだと思った感じが今でも残っている。第一毛をもって装飾されべきはずの顔がつるつるしてまるで薬缶だ。その後猫にもだいぶ逢ったがこんな片輪には一度も出会わした事がない。のみならず顔の真中があまりに突起している。そうしてその穴の中から時々ぷうぷうと煙を吹く。どうも咽せぽくて実に弱った。これが人間の飲む煙草というものである事はようやくこの頃知った。";
```

この文章で使われている文字一覧は，既に使ったことのある`Characters`と`Union`を使うことで，次のように*Mathematica*に調べさせることが出来ます．直前の出力を表すパーセント記号（`%`）も使って，使われている文字の種類（今回の場合，文字一覧のリストに含まれる要素の個数）がいくつあるかも調べさせています．

```
使用文字の一覧 = Union[Characters[吾輩は猫である]]
```

```
{。，々，あ，い，う，え，か，が，き，く，け，げ，こ，
 さ，し，じ，ず，せ，そ，た，だ，ち，っ，つ，て，で，
 と，ど，な，に，ぬ，の，は，ば，ぶ，ぷ，べ，ぽ，ま，み，
 む，め，も，や，ゆ，よ，ら，り，る，れ，ろ，わ，を，ん，
 ス，ニ，フ，ャ，ワ，一，一，上，中，事，人，今，会，何，
 出，別，前，名，吹，吾，咽，妙，始，実，少，度，弱，当，
 彼，後，思，恐，悪，感，憶，我，所，持，捕，掌，族，時，
 暗，書，残，段，毛，泣，無，煙，煮，片，猫，獰，生，番，
 真，知，種，穴，突，第，缶，考，聞，草，落，薄，薬，装，
 見，記，話，起，載，輩，輪，逢，間，頃，顔，食，飲，飾}
```

```
Length[%]
```
```
138
```

文章の文字数は，次のように448文字ですから，かなり同じ文字（そのほとんどは平仮名と思います）が繰り返して使われていることがわかります．Lengthと StringLengthの使い分けをきちんと出来るか大丈夫でしょうか．Lengthはリストに含まれる要素の個数で，StringLengthは文字列の文字の長さです．

```
StringLength[吾輩は猫である]
```
```
448
```

【青空文庫】本文中の「吾輩は猫である」は，夏目漱石全集を底本とする青空文庫のXHTMLファイルから引用しています．引用にあたり，括弧書きのルビは削除してあります．この場をかりて，青空文庫のような素晴らしい取り組みをされている方々に感謝します．貴重な電子データをありがとうございます！

作品に使われている文字の種類を調べてみよう

もっと詳しく調べていくことにしましょう．まずは文字を種類ごとにリストとして定義しましょう．ひらがなとカタカナは以前の通り，次のように定義します（結果を確認する必要がない場合はセミコロンを付けましょう）．

```
ひらがな = CharacterRange["あ", "ん"];
```

```
カタカナ = CharacterRange["ァ", "ヴ"];
```

記号は非常に多くの種類があるので，いくつか主要なものだけを含めて「漢字以外」というグループも定義しておきます．ここでは全角スペース（　）からカタカナの伸ばす記号（ー）までとして定義しています．どのような内容が含まれるかを確認したい場合は，次のようにセミコロンを付けずに実行しましょう．文字が表示されていないところは，機種依存文字であったり，もともと文字が未定義になっている区画になります．

```
漢字以外 = CharacterRange["　", "一"]
```

```
{　, 、, 。, 〃, ㋀, 々, 〆, 〇, 〈, 〉, 《, 》, 「, 」, 『,
 』, 【, 】, 〒, 〓, 〔, 〕, 〖, 〗, 〘, 〙, 〚, 〛, 〜, 〝,
 〞, 〟, 〶, 〳, 〴, 〵, 〻, 〼, 〽, 〾, 〿, ぀, ぁ, あ,
 ぃ, い, ぅ, う, ぇ, え, ぉ, お, か, が, き, ぎ, く, ぐ,
 け, げ, こ, ご, さ, ざ, し, じ, す, ず, せ, ぜ, そ, ぞ,
 た, だ, ち, ぢ, っ, つ, づ, て, で, と, ど, な, に, ぬ,
 ね, の, は, ば, ぱ, ひ, び, ぴ, ふ, ぶ, ぷ, へ, べ, ぺ,
 ほ, ぼ, ぽ, ま, み, む, め, も, ゃ, や, ゅ, ゆ, ょ, よ,
 ら, り, る, れ, ろ, ゎ, わ, ゐ, ゑ, を, ん, ゔ, ゕ, ゖ,
 ゙, ゚, ゛, ゜, ゝ, ゞ, ゟ, ゠, ァ, ア, ィ, イ, ゥ, ウ,
 ェ, エ, ォ, オ, カ, ガ, キ, ギ, ク, グ, ケ, ゲ, コ, ゴ, サ,
 ザ, シ, ジ, ス, ズ, セ, ゼ, ソ, ゾ, タ, ダ, チ, ヂ, ッ,
 ツ, ヅ, テ, デ, ト, ド, ナ, ニ, ヌ, ネ, ノ, ハ, バ, パ,
 ヒ, ビ, ピ, フ, ブ, プ, ヘ, ベ, ペ, ホ, ボ, ポ, マ, ミ,
 ム, メ, モ, ャ, ヤ, ュ, ユ, ョ, ヨ, ラ, リ, ル, レ, ロ,
 ヮ, ワ, ヰ, ヱ, ヲ, ン, ヴ, ヵ, ヶ, ヷ, ヸ, ヹ, ・, ー}
```

グループについての章で使った，Intersection，Complement，Unionなどの組込み関数を使うことで，文章で使われている文字を種類ごとに分類することが出来ます．復讐しておくと，共通部分がIntersectionで，補集合（指定したもの以外の集合）がComplementで，和集合がUnionです．

従って，吾輩は猫であるの冒頭の一段落に含まれるひらがなの種類は次のように命令します．Lengthと%を使うことで，使われているひらがなの種類が52個であることもわかります．

```
Intersection[使用文字の一覧, ひらがな]
```

```
{あ, い, う, え, か, が, き, く, け, げ, こ, さ, し,
 じ, ず, せ, そ, た, だ, ち, っ, つ, て, で, と, ど,
 な, に, ぬ, の, は, ば, ぶ, ぷ, べ, ぽ, ま, み, む,
 め, も, や, ゆ, よ, ら, り, る, れ, ろ, わ, を, ん}
```

```
Length[%]
```
```
52
```

カタカナについても同様です.

```
Intersection[使用文字の一覧, カタカナ]
```
```
{ス, ニ, フ, ャ, ワ}
```

使われている漢字を調べるには，先ほど準備した漢字以外の補集合（漢字以外に含まれていない要素の集合）を*Mathematica*に調べてもらいます．漢字以外には，ひらがなとカタカナ，いくつかの記号が含まれているので，次のように文章に使われている漢字だけを含むリストが得られます．冒頭の段落だけなのに，78種類もの漢字が使われているとは驚きですね.

```
Complement[使用文字の一覧, 漢字以外]
```
```
{一, 上, 中, 事, 人, 今, 会, 何, 出, 別, 前, 名, 吹,
 吾, 咽, 妙, 始, 実, 少, 度, 弱, 当, 彼, 後, 思, 恐,
 悪, 感, 憶, 我, 所, 持, 捕, 掌, 族, 時, 暗, 書, 残,
 段, 毛, 泣, 無, 煙, 煮, 片, 猫, 獰, 生, 番, 真, 知,
 種, 穴, 突, 第, 缶, 考, 聞, 草, 落, 薄, 薬, 装, 見,
 記, 話, 起, 載, 輩, 輪, 逢, 間, 頃, 顔, 食, 飲, 飾}
```

```
Length[%]
```
```
78
```

記号を調べさせるときの命令は少し複雑になります．まず，ひらがなとカタカナと記号だけを使われている文字一覧から抜き出してから，ひらがなとカタカナを取り除きます．`Intersection`と`Complement`を組み合わしているので少し難しいかもしれません.

```
Complement[Intersection[使用文字の一覧, 漢字以外],
 ひらがな, カタカナ]
```

```
{。, 々, ー}
```

「Complement」という命令は正確には「最初のリストから，二番目以降のリストに含まれている要素を全て取り除け」という意味になるので，共通部分を抜き出しているところを分解してみれば難しくないことがわかると思います．

```
使用文字の一覧のうち記号とひらがなとカタカナ =
  Intersection[使用文字の一覧, 漢字以外];
```

```
Complement[使用文字の一覧のうち記号とひらがなとカタカナ,
 ひらがな, カタカナ]
```

```
{。, 々, ー}
```

作品を小学生のつもりになって読んでみよう

以前に小学校第一学年の漢字リストを作ったように，文部科学省のウェブサイトの小学校学習指導要領の国語に関するところに掲載されている「学年別漢字配当表」から，各学年に配当されている漢字のリストを作っておきます．

```
小学校1学年 =
  Characters[
   "一右雨円王音下火花貝学気九休玉金空月犬見五口校左三山子四;
      糸字耳七車手十出女小上森人水正生青夕石赤千川先早草;
      足村大男竹中虫町天田土二日入年白八百文木本名目立力;
      林六"];
```

```
小学校2学年 =
  Characters[
   "引羽雲園遠何科夏家歌画回会海絵外角楽活間丸岩顔汽記帰弓牛;
      魚京強教近兄形計元言原戸古午後語工公広交光考行高黄;
      合谷国黒今才細作算止市矢姉思紙寺自時室社弱首秋週春;
      書少場色食心新親図数西声星晴切雪船線前組走多太体台;
      地池知茶昼長鳥朝直通弟店点電刀冬当東答頭同道読内南;
      肉馬売買麦半番父風分聞米歩母方北毎妹万明鳴毛門夜野;
      友用曜来里理話"];
```

```
小学校3学年 =
  Characters[
   "悪安暗医委意育員院飲運泳駅央横屋温化荷界開階寒感漢館岸起;
      期客究急級宮球去橋業曲局銀区苦具君係軽血決研県庫湖;
      向幸港号根祭皿仕死使始指歯詩次事持式実写者主守取酒;
      受州拾終習集住重宿所暑助昭消商章勝乗植申身神真深進;
      世整昔全相送想息速族他打対待代第題炭短談着注柱丁帳;
      調追定庭笛鉄転都度投豆島湯登等動童農波配倍箱畑発反;
      坂板皮悲美鼻筆氷表秒病品負部服福物平返勉放味命面問;
      役薬由油有遊予羊洋葉陽様落流旅両緑礼列練路和"];
```

```
小学校4学年 =
  Characters[
   "愛案以衣位囲胃印英栄塩億加果貨課芽改械害街各覚完官管関観;
      願希季紀喜旗器機議求泣救給挙漁共協鏡競極訓軍郡径型;
      景芸欠結建健験固功好候航康告差菜最材昨札刷殺察参産;
      散残士氏史司試児治辞失借種周祝順初松笑唱焼象照賞臣;
      信成省清静席積折節説浅戦選然争倉巣束側続卒孫帯隊達;
      単置仲貯兆腸低底停的典伝徒努灯堂働特得毒熱念敗梅博;
      飯飛費必票標不夫付府副粉兵別辺変便包法望牧末満未脈;
      民無約勇要養浴利陸良料量輪類令冷例歴連老労録"];
```

```
小学校5学年 =
  Characters[
   "圧移因永営衛易益液演応往桜恩可仮価河過賀快解格確額刊幹慣;
      眼基寄規技義逆久旧居許境均禁句群経潔件券険検限現減;
      故個護効厚耕鉱構興講混査再災妻採際在財罪雑酸賛支志;
      枝師資飼示似識質舎謝授修述術準序招承証条状常情織職;
      制性政勢精製税責績接設舌絶銭祖素総造像増則測属率損;
      退貸態団断築張提程適敵統銅導徳独任燃能破犯判版比肥;
      非備俵評貧布婦富武復複仏編弁保墓報豊防貿暴務夢迷綿;
      輸余預容略留領"];
```

```
小学校6学年 =
  Characters[
   "異遺域宇映延沿我灰拡革閣割株干巻看簡危机揮貴疑吸供胸郷勤;
      筋系敬警劇激穴絹権憲源厳己呼誤后孝皇紅降鋼刻穀骨困;
      砂座済裁策冊蚕至私姿視詞誌磁射捨尺若樹収宗就衆従縦;
      縮熟純処署諸除将傷障城蒸針仁垂推寸盛聖誠宣専泉洗染;
      善奏窓創装層操蔵臓存尊宅担探誕段暖値宙忠著庁頂潮賃;
      痛展討党糖届難乳認納脳派拝背肺俳班晩否批秘腹奮並陛;
      閉片補暮宝訪亡忘棒枚幕密盟模訳郵優幼欲翌乱卵覧裏律;
      臨朗論"];
```

面倒なので，小学校の各学年に配当された全ての漢字を含むリスト「小学校配当漢字」も定義しておきます．ここでは「複数のリストをひとつのリストにまとめなさい」という意味を持つ「Join」という命令を使っていますが，Unionでも同じ結果になります．Unionでは，まとめると共に重複要素があれば削除しなさい，という補足条件が付いてきます．

```
小学校配当漢字 = Join[小学校1学年, 小学校2学年,
    小学校3学年, 小学校4学年, 小学校5学年, 小学校6学年];
```

前回に使った「使用文字の一覧」と「漢字以外」を組み合わせることで，次のように小学校配当漢字に含まれない漢字のみを抜き出すことが出来ます（これを未配当漢字として定義しておきましょう）．そんなに難しそうな漢字を含んでいるようには見えませんし，わずか一段落ですが，21種類もの小学校に配当されていない漢字を含んでいることがわかります．

```
未配当漢字 = Complement[使用文字の一覧, 漢字以外,
   小学校配当漢字]
```

```
{吹, 吾, 咽, 妙, 彼, 恐, 憶, 捕, 掌, 煙,
 煮, 猫, 獰, 突, 缶, 薄, 載, 輩, 逢, 頃, 飾}
```

```
Length[%]
```

```
21
```

早口言葉のところで何度も使ったStringReplaceを使うと，未配当漢字を□マークなどに置き換えることで，小学生になった気分で作品を読むことが出来ます．次の例は実際に「猫」という文字を「□」に置き換えてみたものです．出力された作品中で「猫」が「□」に変わっているのがわかると思います．

```
StringReplace[吾輩は猫である, "猫" → "□"]
```

```
吾輩は□である。名前はまだ無い。どこで生れたかとんと見当がつか
    ぬ。何でも薄暗いじめじめした所でニャーニャー泣いていた事だ
    けは記憶している。吾輩はここで始めて人間というものを見た。
    しかもあとで聞くとそれは書生という人間中で一番獰悪な種族で
    あったそうだ。この書生というのは時々我々を捕えて煮て食うと
    いう話である。しかしその当時は何という考もなかったから別段
    恐しいとも思わなかった。ただ彼の掌に載せられてスーと持ち上
    げられた時何だかフワフワした感じがあったばかりである。掌の
    上で少し落ちついて書生の顔を見たのがいわゆる人間というもの
    の見始であろう。この時妙なものだと思った感じが今でも残って
    いる。第一毛をもって装飾されべきはずの顔がつるつるしてまる
    で薬缶だ。その後□にもだいぶ逢ったがこんな片輪には一度も出
    会わした事がない。のみならず顔の真中があまりに突起している
    。そうしてその穴の中から時々ぷうぷうと煙を吹く。どうも咽せ
    ぽくて実に弱った。これが人間の飲む煙草というものである事は
    ようやくこの頃知った。
```

この作業を全ての未配当漢字に対して繰り返せば，確かに小学生の気持ちになって作品を読むことは出来ます．しかしながら，余りにも冗長で計算

機のパワーを，即ち*Mathematica*のパワーを使っていません．冗長な作業は*Mathematica*に任せることが重要です．

*Mathematica*における文字の置き換え規則には，「リストの中のどれか」という意味を持つ「Alternatives」という命令が使えます．従って，次のような置き換え規則を使うことで，未配当漢字の全てを「□」に置き換えることが出来ます．当り前のことかもしれませんが，小学生に「吾輩は猫である」は少し難しいかもしれないことがわかります．

```
StringReplace[吾輩は猫である，
 Alternatives[未配当漢字] → "□"]
```

□□は□である。名前はまだ無い。どこで生れたかとんと見当がつかぬ。何でも□暗いじめじめした所でニャーニャー泣いていた事だけは記□している。□□はここで始めて人間というものを見た。しかもあとで聞くとそれは書生という人間中で一番□悪な種族であったそうだ。この書生というのは時々我々を□えて□て食うという話である。しかしその当時は何という考もなかったから別段□しいとも思わなかった。ただ□の□に□せられてスーと持ち上げられた時何だかフワフワした感じがあったばかりである。□の上で少し落ちついて書生の顔を見たのがいわゆる人間というものの見始であろう。この時□なものだと思った感じが今でも残っている。第一毛をもって装□されべきはずの顔がつるつるしてまるで薬□だ。その後□にもだいぶ□ったがこんな片輪には一度も出会わした事がない。のみならず顔の真中があまりに□起している。そうしてその穴の中から時々ぷうぷうと□を□く。どうも□せぽくて実に弱った。これが人間の飲む□草というものである事はようやくこの□知った。

学年別配当漢字を使って細かく調べてみよう

この作品をもう少し詳しく調べてみることにしましょう．*Mathematica*に調べさせるのに必要な組込み関数は既に使ったものばかりです．文字列の長さを調べさせるStringLength，該当する文字の個数を調べさせるStringCount，リストの要素のどれかに該当という意味を持つAlternativesです．まずは，対象の文章の長さを確認しておきましょう．

```
文章の長さ = StringLength[吾輩は猫である]
```

```
448
```

小学校の各学年に配当されている漢字の個数を確認していきましょう．StringCountとAlternativesで使われている漢字の個数を求められます．各学年ごとに6回繰り返すことで，それぞれの配当漢字の個数がわかります．

```
StringCount[吾輩は猫である, Alternatives[小学校1学年]]
```

```
23
```

```
StringCount[吾輩は猫である, Alternatives[小学校2学年]]
```

```
36
```

```
StringCount[吾輩は猫である, Alternatives[小学校3学年]]
```

```
20
```

```
StringCount[吾輩は猫である, Alternatives[小学校4学年]]
```

```
6
```

```
StringCount[吾輩は猫である, Alternatives[小学校5学年]]
```

```
0
```

```
StringCount[吾輩は猫である, Alternatives[小学校6学年]]
```

```
5
```

文字が赤くなっていますが，この赤字で強調表示している部分は，それぞれの命令に共通している部分です．変化しているのは学年別の配当漢字を覚えさせたリストの名前だけです．このように共通する部分が多い場合，以前にも出てきましたが．これと同じ指示を組込み関数「Map」を使う

と簡単に書くことが出来ます．この命令の意味は「次のリストのそれぞれに対して同じ命令を実行せよ」です．

今回の場合，以前に比べて共通する部分が多いので，次のように少し複雑になります．赤い文字の部分は，個別に命令した場合に共通していた部分を表しています．共通しなかった部分が「シャープ（#）」になり，最後に「アンパサンド（&）」が付け加わっているのが変更部分です．以前のMapの例では，#と&が省略されていたのですが，Mapを使うときは基本的に，共通しない部分を#に，共通する部分の最後に&を付け加えなければいけません．

```
Map[StringCount[吾輩は猫である, Alternatives[#]] &,
 {小学校1学年, 小学校2学年, 小学校3学年, 小学校4学年,
  小学校5学年, 小学校6学年}]
```

```
{23, 36, 20, 6, 0, 5}
```

リストの一括処理：いくつものリストに同じ操作を実行

ひらがなとカタカナの個数も同時に調べるには，次のように，ひらがなとカタカナも追加するだけで，300個と9個ということがわかります．

```
Map[StringCount[吾輩は猫である, Alternatives[#]] &,
 {小学校1学年, 小学校2学年, 小学校3学年, 小学校4学年,
  小学校5学年, 小学校6学年, ひらがな, カタカナ}]
```

```
{23, 36, 20, 6, 0, 5, 300, 9}
```

```
%/文章の長さ*100.0
```

```
{5.13393, 8.03571, 4.46429,
 1.33929, 0, 1.11607, 66.9643, 2.00893}
```

次章で扱いますが，求めた文字の個数を「文章の長さ」で割って100倍すれば使用率がパーセントで求められます．わり算は「スラッシュ

（/）」で，かけ算は「アスタリスク（*）」で命令します．これによれば，小学校1学年の配当漢字は5%，2学年が8%，3学年が4%，4学年が1%，5学年が0%，6学年が1%，ひらがなが67%，カタカナが2%使われていることがわかります．

四則演算で面白い数を見付けてみよう

これまでMathematicaにリストや文字列などの数学とはかけ離れた内容を扱わせてきましたが，Mathematicaには難しい計算をさせることも出来ます．ここでは，基本的な四則演算（足す，引く，掛ける，割る）について取り上げていきたいと思います．

足し算（+）で足して10になる数を探そう

Mathematicaに足し算を行わせるには，算数や数学でも使われるプラス記号「+」を使います．教科書などにあるように，そのまま命令として入力すれば，Mathematicaは計算結果を教えてくれます．例えば，単純な「1+2+3+4」もそのまま入力するだけで十分です．答えが「10」であると教えてくれます．

```
1 + 2 + 3 + 4
```

```
10
```

これまで組込み関数と呼ばれるMathematicaへ指示するための命令をいくつも習ってきました．実は，足し算にもプラス記号とは別に，「次の数を足し合わせよ」という意味を持つ「Plus」という命令も用意されています．例えば，この命令を使って先ほどの単純な「1+2+3+4」を計算するには次のようにします．もちろん，答えは「10」になります．

```
Plus[1, 2, 3, 4]
```

```
10
```

では，日本で余り見ないタイプの問題「3+□=10」を考えましょう．日常的には良くある状況を表していて，例えば，パーティの参加者は10人なのにコップが3人分しかない，あといくつコップを用意すれば良いでしょうか，という問題になります．

普通の考え方とは違いますが，用意しなければならないコップの数を直接求めるのではなく，新たにいくつコップを追加すると10人分になるのかを考えましょう．そこで，追加するコップの個数とし

て，5個，6個，7個，8個，9個，10個の場合を考えます．分かりやくするために，追加する可能性のあるコップの個数を*Mathematica*に覚えさせておきます．

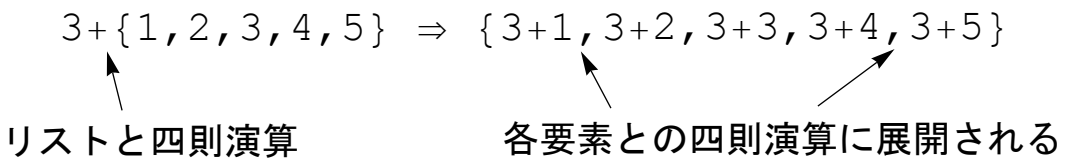
```
追加するコップの個数 = {5, 6, 7, 8, 9, 10}
```
```
{5, 6, 7, 8, 9, 10}
```

問題は，既にある3個のコップに加えると10個になるコップの個数ですから，この数のそれぞれに3を加えて10になるかを確認すれば良いですね．プラス記号「+」を使って確認しようとすると，6回も足し算を命令する必要があります．非常に面倒ですね．*Mathematica*では面倒なことをしなくても良いように，もっと便利な方法を用意しています．それは．．．

『リストに足し算を行ってしまう』

です．*Mathematica*では「数とリストの足し算」を，「リストの各要素ごとに同じ足し算をしたいのだろう」と解釈してくれます．従って，次のように単純に3をリストに加えることで一度に全ての足し算を指示することが出来ます．

```
3 + 追加するコップの個数
```
```
{8, 9, 10, 11, 12, 13}
```



リストと四則演算：各要素に対しての同じ四則演算が行われる

実際には次のように，以前に出てきたTableFormをオプション（追加指示）TableHeadingsと組み合わせて使うことで，よりわかりやすい見た目となります．

```
TableForm[{追加するコップの個数, 3 + 追加するコップの個数},
 TableHeadings → {{"追加するコップの個数", "合計"}, None}]
```

追加するコップの個数	5	6	7	8	9	10
合計	8	9	10	11	12	13

もちろん，同じ操作を何度も繰り返す，という視点からは文字列操作で出てきたMapを使うことも出来ますね．足し算の2種類の書き方のそれぞれに対して，実際にMapを使ってみたものが次になります．直接リストと足し算した方が簡単で便利だということがわかると思います．

```
Map[Plus[3, #] &, 追加するコップの個数]
```

```
{8, 9, 10, 11, 12, 13}
```

```
Map[(3 + #) &, 追加するコップの個数]
```

```
{8, 9, 10, 11, 12, 13}
```

引き算（-）で足して10になることを確認しよう

*Mathematica*に引き算を行わせるには，算数や数学でも使われるマイナス記号「-」を使います．教科書などにあるように，そのまま命令として入力すれば，*Mathematica*は計算結果を教えてくれます．例えば，単純な「20-4-3-2-1」もそのまま入力するだけで十分です．答えが「10」であると教えてくれます．

```
20 - 4 - 3 - 2 - 1
```

```
10
```

引き算にも，足し算と同じように，「次の2数の引き算をせよ」という意味を持つ「Subtract」という命令も用意されています．ただし，足し算と異なり2数に対してしか使えません．例えば，この命令を使って単純な「20-10」を計算するには次のようにします．もちろん，答えは「10」になります．

```
Subtract[20, 10]
```

```
10
```

では，先ほどと同じ問題「3+□=10」を考えましょう．これを「10-□=3」と変形して解きましょう．即ち，10から何を引いたら3になるのか，を考えましょう．そこで，引くコップの個数として，5個，6個，7個，8個，9個，10個の場合を考えます．先ほど同じように，引き算するコップの個数を*Mathematica*に覚えさせておきます．

```
引くコップの個数 = {5, 6, 7, 8, 9, 10}
```

```
{5, 6, 7, 8, 9, 10}
```

引き算についても*Mathematica*では「数とリストの引き算」を，「リストの各要素ごとに同じ引き算をしたいのだろう」と解釈してくれます．従って，次のように単純に10からリストを引くことで一度に全ての引き算を指示することが出来ます．

```
10 - 引くコップの個数
```

```
{5, 4, 3, 2, 1, 0}
```

実際には次のように，以前に出てきたTableFormをオプション（追加指示）TableHeadingsと組み合わせて使うことで，よりわかりやすい見た目となります．

```
TableForm[{引くコップの個数, 10 - 引くコップの個数},
 TableHeadings → {{"引くコップの個数", "残り"}, None}]
```

引くコップの個数	5	6	7	8	9	10
残り	5	4	3	2	1	0

もちろん，同じくMapを使うことも出来ます．

```
Map[Subtract[10, #] &, 引くコップの個数]
```

```
{5, 4, 3, 2, 1, 0}
```

```
Map[(10 - #) &, 引くコップの個数]
```

```
{5, 4, 3, 2, 1, 0}
```

【数学的表記と四則演算】*Mathematica*での計算に関する指示は，なるべく算数や数学の慣用的な表現方法を踏襲するようになっています．従って，教科書等にある計算問題はそのまま入力すれば，結果を*Mathematica*が求めて教えてくれます．実際に，この項では足し算と引き算を習っていますが，他の演算や括弧なども含め，慣用的な表現がそのまま利用できるようになっています．

掛け算（×）して規則正しい数を探してみよう

*Mathematica*に掛け算を行わせるには，算数や数学でも使われる記号「×」を使います．教科書などにあるように，そのまま命令として入力すれば，*Mathematica*は計算結果を教えてくれます．例えば，単純な「1×2×3×4」もそのまま入力するだけで十分です．答えが「24」であると教えてくれます．なお，記号「×」は「ESC*ESC」で入力できます．

```
1×2×3×4
```

```
24
```

かけ算の記号には，「×」の他，アスタリスク「*」やスペース「 」も使えます．このうちスペースについては，*Mathematica*が自動的に「×」を補って表示してくれますので，入力簡単で表示もわかりやすいのでお薦めです．

```
1*2*3*4
```

```
24
```

```
1×2×3×4
```

```
24
```

更に掛け算にも，足し算や引き算と同じように，「次の数を掛け合わせよ」という意味を持つ「Times」という命令も用意されています．例え

ば，この命令を使って先ほどの単純な「1×2×3×4」を計算するには次のようにします．もちろん，答えは「24」になります．

```
Times[1, 2, 3, 4]
```
```
24
```

次の2つの掛け算を見てください．

```
110 011 * 101
```
```
11 111 111
```

```
110 011 * 202
```
```
22 222 222
```

*Mathematica*では算数や数学で習う計算順序（掛け算を足し算や引き算よりも先に行う）に従って計算をします．従って，何も考えずに教科書にあるような計算式をそのまま指示するだけで答えが得られます．

```
110 011 * 202 - 110 011 * 101
```
```
11 111 111
```

また，括弧による計算順序の指定も可能です．このとき使う括弧は必ず丸括弧（「(」と「)」）にしてください．算数や数学で使用することもある「{}」や「[]」は，*Mathematica*では別のに意味で使われますので，何重になっても計算順序の指定には丸括弧だけを使います．

```
(10 000 + 100 + 1) * 11
```
```
111 111
```

```
10 101 * 11
```
```
111 111
```

足し算や引き算と同じく，*Mathematica*では「数とリストの掛け算」を，「リストの各要素ごとに同じ掛け算をしたいのだろう」と解釈してくれます．この機能をうまく使って，「12345679」に何を掛けたら

「111111」のような規則正しい数になるかを考えましょう．試しに，1から5のリストと積をとってみます．規則正しそうな数は表れません．

```
12 345 679 * {1, 2, 3, 4, 5}
```

```
{12 345 679, 24 691 358, 37 037 037, 49 382 716, 61 728 395}
```

もっと大きな数，もっとたくさんの数との積を計算させないと見付かりそうにありません．でも，そんなに長いリストを手で入力するのは面倒です．*Mathematica*には「次の数までの整数のリストを作成せよ」という意味の「Range」なる命令があります．これを使うことで，次のように簡単にリストを作ることが出来ます．

```
Range[20]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
```

従って，次のように使うことで，9のときと18のときに「111111」のような規則正しい数になることがわかります．

```
12 345 679 * Range[20]
```

```
{12 345 679, 24 691 358, 37 037 037, 49 382 716,
 61 728 395, 74 074 074, 86 419 753, 98 765 432,
 111 111 111, 123 456 790, 135 802 469, 148 148 148,
 160 493 827, 172 839 506, 185 185 185, 197 530 864,
 209 876 543, 222 222 222, 234 567 901, 246 913 580}
```

TableFormに「行方向に表形式で表示せよ」というオプション「TableDirections → Row」を付けて指示を出すことで，次のようにわかりやすい結果も得られます．

```
TableForm[{Range[20], 12 345 679 * Range[20]},
 TableDirections → Row]
```

```
1   12 345 679
2   24 691 358
3   37 037 037
4   49 382 716
5   61 728 395
6   74 074 074
7   86 419 753
8   98 765 432
9   111 111 111
10  123 456 790
11  135 802 469
12  148 148 148
13  160 493 827
14  172 839 506
15  185 185 185
16  197 530 864
17  209 876 543
18  222 222 222
19  234 567 901
20  246 913 580
```

【数のブロック表示】*Mathematica*では次のように比較的大きな数を入力すると，3桁ずつまとめて表示を行います．日本式では4桁ずつの方がわかりやすいかもしれません．4桁ずつに変更したい場合は，編集メニューの「環境設定...」で表示されるダイアログにて，「外観」のタブの中の「数」のタブの中の「桁ブロックのサイズ」を変更してください．このダイアログでは様々な設定を変更できるようになっています．いろいろと変更してみるのも面白いかもしれません．「デフォルト値に戻す」ボタンで元に戻せるので安心して変更してみてください．

▫ 割り算（/）も使って分数の計算をしてみよう

*Mathematica*に割り算を行わせるには，算数や数学でも使われる記号「÷」を使います．教科書などにあるように，そのまま命令として入力すれば，*Mathematica*は計算結果を教えてくれます．例えば，単純な「20÷2」もそのまま入力するだけで十分です．答えが「10」であると教えてくれます．なお，記号「÷」は「ESC div ESC」で入力できます．

```
20 ÷ 2
```

```
10
```

割り算の記号には，「÷」の他，スラッシュ「/」も使えます．

```
20 / 2
```

```
10
```

更に割り算にも，他の四則演算と同じように，「次の2数の割り算を求めよ」という意味を持つ「Divide」という命令も用意されています．例えば，この命令を使って先ほどの単純な「20÷2」を計算するには次のようにします．もちろん，答えは「10」になります．

```
Divide[20, 2]
```

```
10
```

割り切れない場合，*Mathematica*は結果を分数で教えてくれます．

```
1 / 2
```

$$\frac{1}{2}$$

他の四則演算と同じく，*Mathematica*では「数とリストの割り算」を，「リストの各要素ごとに同じ割り算をしたいのだろう」と解釈してくれます．この機能をうまく使って，「3」で割り切れる数字を1から20までの整数から探してみましょう．

```
Range[20] / 3
```

$$\left\{\frac{1}{3}, \frac{2}{3}, 1, \frac{4}{3}, \frac{5}{3}, 2, \frac{7}{3}, \frac{8}{3}, 3, \frac{10}{3}, \frac{11}{3}, 4, \frac{13}{3}, \frac{14}{3}, 5, \frac{16}{3}, \frac{17}{3}, 6, \frac{19}{3}, \frac{20}{3}\right\}$$

```
TableForm[{Range[20], Range[20] / 3}]
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$\frac{1}{3}$	$\frac{2}{3}$	1	$\frac{4}{3}$	$\frac{5}{3}$	2	$\frac{7}{3}$	$\frac{8}{3}$	3	$\frac{10}{3}$	$\frac{11}{3}$	4	$\frac{13}{3}$	$\frac{14}{3}$	5	$\frac{16}{3}$	$\frac{17}{3}$	6	$\frac{19}{3}$

割り算の順番を変えることで，「30」を割り切る数を1から20までの整数から探すことも出来ます．

```
TableForm[{Range[20], 30 / Range[20]}]
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	1
30	15	10	$\frac{15}{2}$	6	5	$\frac{30}{7}$	$\frac{15}{4}$	$\frac{10}{3}$	3	$\frac{30}{11}$	$\frac{5}{2}$	$\frac{30}{13}$	$\frac{15}{7}$	2	$\frac{15}{8}$	$\frac{30}{17}$	

分数同士の四則演算もそのまま行えます．

```
1 / 2 + 1 / 3
```

$$\frac{5}{6}$$

より分数らしく入力したい場合は，CTRLを押しながら「/」を入力します．例えば，次のような分数の計算式を入力するには，「1」を入力，CTRLを押しながら「/」を入力，「2」を入力，カーソルキーの右，「+」を入力，「1」を入力，CTRLを押しながら「/」を入力，「3」を入力します．カーソルキーの右は，CTRLを押しながらスペースを入力でも代用できます．

$$\frac{1}{2} + \frac{1}{3}$$

$$\frac{5}{6}$$

【帯分数と仮分数】*Mathematica*は分数を仮分数で表現します．帯分数では掛け算との区別が難しいためです．区別が難しいため利用はお薦めしませんが，どうしても帯分数で入力を行いたい場合は，暗黙の加算である「ESC+ESC」を使ってください．「1ESC+ESC$\frac{1}{3}$」で「$1\frac{1}{3}$」と入力でき，「$1\frac{1}{3}$」を表します．

累乗（^）を使って複利計算をしてみよう

*Mathematica*に累乗（冪乗，指数計算）を行わせるには，算数や数学での表現とは異なる記号「^」を使います．教科書などにあるような表現（数の右肩に小さく指数を書く方法）にしたい場合は，パレット（□□）を使うか後述するキーボードショートカットを利用してください．例えば，記号「^」を使う場合，単純な「2^{10}」を入力するには「2^10」と入力します．答えが「1024」であると教えてくれます．

```
2^10
```

```
1024
```

累乗にも四則演算（加減乗除）と同じように，「次の累乗を求めよ」という意味の「Power」という命令も用意されています．例えば，この命令を使って先ほどの単純な「2^10」を計算するには次のようにします．もちろん，答えは「1024」になります．

```
Power[2, 10]
```

```
1024
```

四則演算と同じく，*Mathematica*では「数とリストの累乗」を，「リストの各要素ごとに同じ累乗をしたいのだろう」と解釈してくれます．この機能をうまく使って，「3」を何度掛けたら1000に近くなるか，1から10までの整数から探してみましょう．結果をTableFormで表示すると，3を6回掛けると1000に近いことがわかります．

```
3^Range[10]
```

```
{3, 9, 27, 81, 243, 729, 2187, 6561, 19683, 59049}
```

```
TableForm[{Range[10], 3^Range[10]}]
```

```
1  2  3   4   5    6    7     8     9      10
3  9  27  81  243  729  2187  6561  19683  59049
```

より指数らしく入力したい場合は，CTRLを押しながら「^」を入力します．例えば，次のような累乗の計算式を入力するには，「2」を入力，CTRLを押しながら「^」を入力，「10」を入力，カーソルキーの右，「+」を入力，「3」を入力，CTRLを押しながら「^」を入力，「10」を入力します．カーソルキーの右は，CTRLを押しながらスペースを入力でも代用できます．

```
2^10 + 3^10
```

```
60073
```

累乗の計算が出来ると複利計算が簡単に出来ます．複利計算は，年利を小数表示に直したものに1を加えた数を，年数分だけ累乗するだけです．例えば，0.5%の年利の場合は1.005を年数分だけ累乗すれば良いことになります．10年の場合の計算をしてみます．1.05114ということは，10年間での利率は約5.1%ということになります．

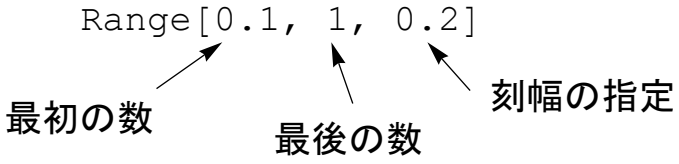
```
1.005^10
```

```
1.05114
```

0.1%から1%まで0.2%刻みの年利で，10年後にどの程度違うかを計算してみましょう．まずは，Rangeを使って年利のリストを作りましょう．Rangeでは次のように使うことで，0.1から1まで0.2刻みの数を作ることが出来ます．

```
年利 = Range[0.1, 1, 0.2]
```

```
{0.1, 0.3, 0.5, 0.7, 0.9}
```



リストの作成：刻幅を指定して数のリストを作る

年利をパーセントから小数に直します．

```
年利の割合 = 年利 / 100
```

```
{0.001, 0.003, 0.005, 0.007, 0.009}
```

年利の割合に1を加えます．1は元本に対応しています．

```
元本込の割合 = 年利の割合 + 1
```

```
{1.001, 1.003, 1.005, 1.007, 1.009}
```

複利計算で10年後の元本比を計算し，計算結果から1を引き，100を掛けて小数からパーセントに直すことで10年間での利率を計算します．結果を見ると，かなり違うことがわかりますね．なお，何ら計算結果を保証するものではありませんので，あしからず．

```
TableForm[{年利, ((元本込の割合^10) - 1) * 100},
 TableHeadings → {{"年利", "10年間での利率"}, None}]
```

年利	0.1	0.3	0.5	0.7	0.9
10年間での利率	1.00451	3.04083	5.11401	7.22467	9.37339

■ 数のリストから目当てのものを捜し出そう

▫ 素数を探そう

素数とは「1とその数自身以外に正の約数を持たない（つまり1とその数以外のどんな自然数によっても割り切れない），1より大きな自然数」のことです．具体的に素数を探してみましょう．*Mathematica*では「n番目の素数を教えなさい」という意味の命令「Prime」を使うことで，簡単に素数を知ることが出来ます．例えば，小さい順に10番目の素数は次のように求められます．

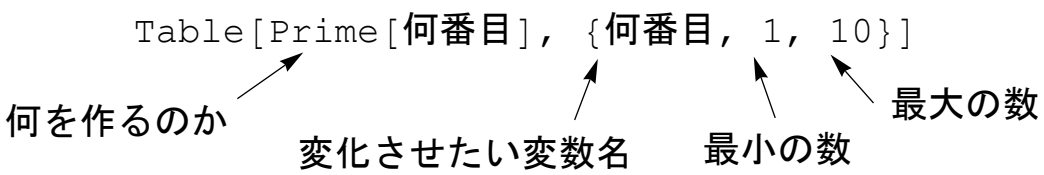
```
Prime[10]
```

```
29
```

作文のときに使った組込み関数のTableを使うことで，より簡単に素数を調べることが出来ます．以前とは異なり，Primeには何番目の素数を知りたいのかを指定する必要があるので，Tableの使い方も以前とは変わってきます．例えば，1番目から10番目までの素数を知りたい場合は，次のように，何番目という変数を1から10まで変化させつつ「Prime[何番目]」を評価することが必要になります．

```
Table[Prime[何番目], {何番目, 1, 10}]
```

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```



リストの作成：同じ仕組みでたくさん作るには

もちろん，いままでに登場した他の組込み関数を使っても同じことは可能です．例えば，次の例はMapとRangeを組み合わせた方法です．

```
Map[Prime, Range[10]]
```

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

```
Map[Prime[#] &, Range[10]]
```

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

*Mathematica*にはもっと便利な方法も用意されています．まずは，Primeという命令の属性を「関数の属性を調べよ」という意味の「Attributes」なる命令で調べましょう．すると，「リストを自動展開して，個々の要素に対して関数を適用する」という属性である「Litable」をPrimeが持っていることがわかります．

```
Attributes[Prime]
```

```
{Listable, Protected}
```

即ち，次のように命令しても，1番目から10番目までの素数を知ることが出

来ます.

```
Prime[Range[10]]
```

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

ところで本当に「29」は素数でしょうか．素数であれば「1とその数以外のどんな自然数によっても割り切れない」はずですから，本当に割り切れないか調べましょう．29よりも大きな数で割り切れないことは明らかですから，1から29までの整数で割ってみましょう．確かに，1と29以外の数では割り切れていません.

```
29 / Range[29]
```

$$\left\{29, \frac{29}{2}, \frac{29}{3}, \frac{29}{4}, \frac{29}{5}, \frac{29}{6}, \frac{29}{7}, \frac{29}{8}, \frac{29}{9}, \frac{29}{10}, \frac{29}{11}, \frac{29}{12}, \frac{29}{13}, \frac{29}{14}, \frac{29}{15}, \frac{29}{16}, \frac{29}{17}, \frac{29}{18}, \frac{29}{19}, \frac{29}{20}, \frac{29}{21}, \frac{29}{22}, \frac{29}{23}, \frac{29}{24}, \frac{29}{25}, \frac{29}{26}, \frac{29}{27}, \frac{29}{28}, 1\right\}$$

では「14351」はどうでしょうか．全部で14351個も数が表示されることになるので，*Mathematica*が気を利かせて，一部分だけの出力をしてくれています．表示を増やしながら割り切れているかを確認しても良いのですが，非常に面倒です.

```
14 351 / Range[14 351]
```

非常に大きな出力が生成されました．以下はそのサンプルです.

$$\left\{14\,351, \frac{14\,351}{2}, \frac{14\,351}{3}, \frac{14\,351}{4}, \frac{14\,351}{5}, \frac{14\,351}{6}, \frac{14\,351}{7}, \frac{14\,351}{8}, \frac{14\,351}{9}, \frac{14\,351}{10}, \ll 14\,331 \gg, \frac{14\,351}{14\,342}, \frac{14\,351}{14\,343}, \frac{14\,351}{14\,344}, \frac{14\,351}{14\,345}, \frac{14\,351}{14\,346}, \frac{14\,351}{14\,347}, \frac{14\,351}{14\,348}, \frac{14\,351}{14\,349}, \frac{14\,351}{14\,350}, 1\right\}$$

表示を少なく | もっと表示

全出力を表示 | 大きさ制限の設定...

このような場合は，「14351/Range[14351]」の結果であるリストから知りたい情報だけを抜き出して表示させた方が便利です．割り切れていれば整数で，割り切れていなければ分数ですから，整数だけを抜き出すことにしましょう．

整数だけの抜き出しには，「次の条件を満たすものだけをリストから抜き出しなさい」という意味の「Select」なる関数と，「整数かどうか調べなさい」という意味の「IntegerQ」を組み合わせます．結果を見ると，1と14351以外に127と113でも割り切れていますから，14351は素数でないことがわかります．

```
Select[14 351 / Range[14 351], IntegerQ]
```

```
{14 351, 127, 113, 1}
```

Select[調べたいリスト，抽出したい条件]

リストの名称やリストそのもの

主な例としては NumberQ NumericQ

リストからの選出：条件を満たす要素だけを取り出す

条件として使える関数には様々なものがあります．必ずしも使えるとは限りませんが，命令の最後に「Q」が付いているものは，*Mathematica*に何かを問い合わせる命令なので，条件として使えるも野が多くあります．

```
? System`*Q
```

▼ System`

AlgebraicIntegerQ	MatrixQ
AlgebraicUnitQ	MemberQ
ArgumentCountQ	NameQ
ArrayQ	NumberQ
AtomQ	NumericQ
CoprimeQ	OddQ
DigitQ	OptionQ
DistributionDomainQ	OrderedQ
DistributionParameterQ	PartitionsQ
EllipticNomeQ	PolynomialQ
EvenQ	PositiveDefiniteMatrixQ
ExactNumberQ	PossibleZeroQ
FreeQ	PrimePowerQ
HermitianMatrixQ	PrimeQ
HypergeometricPFQ	QuadraticIrrationalQ
InexactNumberQ	RootOfUnityQ
IntegerQ	SameQ
IntervalMemberQ	SquareFreeQ
InverseEllipticNomeQ	StringFreeQ
LegendreQ	StringMatchQ
LetterQ	StringQ
LinkConnectedQ	SyntaxQ
LinkReadyQ	TensorQ
ListQ	TrueQ
LowerCaseQ	UnsameQ
MachineNumberQ	UpperCaseQ
MatchLocalNameQ	ValueQ
MatchQ	VectorQ

双子素数，三つ子素数，四つ子素数を探そう

素数についてもっと調べていきましょう．素数の中には中の良い双子のような数もあり，「差が2の二つの素数の組」のことを双子素数といいます．双子素数の例としては，「3と5」や「11と13」などがあります．

*Mathematica*を使って双子素数を捜し出す方法はいろいろとありますが，ここでは比較的単純で簡単な方法を使ってみます（もっと良い方法もありますが，もっと詳しくなってからにしましょう）．まずは，双子素数の候補となる隣り合った素数のペアを作らなければいけませんが，次のように「素数同士の差」も含めたグループにしておきます．

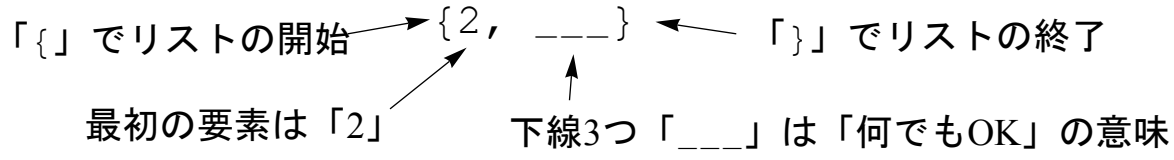
```
差と素数のペア =
 Table[{Prime[n + 1] - Prime[n], Prime[n], Prime[n + 1]},
  {n, 1, 20}]
```

```
{{1, 2, 3}, {2, 3, 5}, {2, 5, 7}, {4, 7, 11},
 {2, 11, 13}, {4, 13, 17}, {2, 17, 19}, {4, 19, 23},
 {6, 23, 29}, {2, 29, 31}, {6, 31, 37}, {4, 37, 41},
 {2, 41, 43}, {4, 43, 47}, {6, 47, 53}, {6, 53, 59},
 {2, 59, 61}, {6, 61, 67}, {4, 67, 71}, {2, 71, 73}}
```

双子素数であれば，赤字の部分で求めさせた「素数同士の差」は2になっているはずです．そこで，「パターンにマッチする要素を取り出しなさい」という意味の「Cases」なる命令で，双子素数を含むグループのみを取り出します．Casesは以前に出てきた文字列用の同じような関数StringCasesの拡張バージョンと思ってください．

```
差と双子素数のグループ = Cases[差と素数のペア, {2, ___}]
```

```
{{2, 3, 5}, {2, 5, 7}, {2, 11, 13}, {2, 17, 19},
 {2, 29, 31}, {2, 41, 43}, {2, 59, 61}, {2, 71, 73}}
```

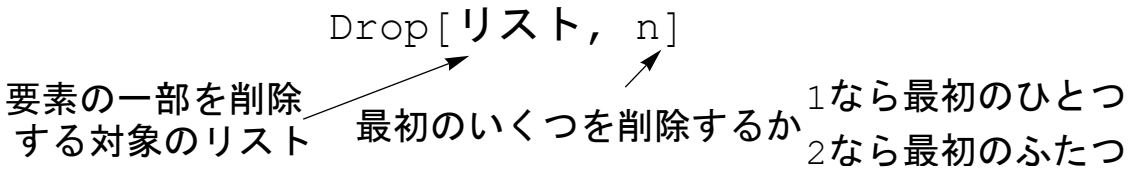


リストのパターン：最初の要素が「2」のリスト

この段階で，Casesにより双子素数を含むグループだけを選別できましたが，いまとなっては余計な「素数同士の差」も含まれてしまっています．これを取り除かないと双子素数のペアにはなりません．要素の削除には「Drop」という命令を使います．この意味は「リストの最初の要素から指定した個数分だけ削除しなさい」です．例えば，「{2,3,5}」から最初の1つを削除し「{3,5}」にするには次のようにします．

```
Drop[{2, 3, 5}, 1]
```

```
{3, 5}
```



リストから要素を削除：最初のいくつかを削除

従って，DropをMapと組み合わせて使うことで，個々のグループから余計な部分を削ぎ落とした，次のような双子素数のペアのリストを捜し出すことが出来ます．この結果を見る限りでは，双子素数はたくさんありますね．

```
Map[Drop[#, 1] &, 差と双子素数のグループ]
```

```
{{3, 5}, {5, 7}, {11, 13}, {17, 19},
 {29, 31}, {41, 43}, {59, 61}, {71, 73}}
```

双子素数を探した方法を少し変えることで，三つ子素数と呼ばれる「差が2である3つの素数の組」も見付けることが出来ます．三つ子素数の例としては「3と5と7」です．双子素数の場合と異なるのは，赤字の部分の差を隣り合ったふたつのペアそれぞれで計算していること，三つ子なので素数のトリオになっていることです．

```
差と素数のトリオ =
 Table[{Prime[n + 2] - Prime[n + 1],
   Prime[n + 1] - Prime[n], Prime[n], Prime[n + 1],
   Prime[n + 2]}, {n, 1, 20}]
```

```
{{2, 1, 2, 3, 5}, {2, 2, 3, 5, 7},
 {4, 2, 5, 7, 11}, {2, 4, 7, 11, 13},
 {4, 2, 11, 13, 17}, {2, 4, 13, 17, 19},
 {4, 2, 17, 19, 23}, {6, 4, 19, 23, 29},
 {2, 6, 23, 29, 31}, {6, 2, 29, 31, 37},
 {4, 6, 31, 37, 41}, {2, 4, 37, 41, 43},
 {4, 2, 41, 43, 47}, {6, 4, 43, 47, 53},
 {6, 6, 47, 53, 59}, {2, 6, 53, 59, 61},
 {6, 2, 59, 61, 67}, {4, 6, 61, 67, 71},
 {2, 4, 67, 71, 73}, {6, 2, 71, 73, 79}}
```

この個々のリストの中から，最初の2つの差がどちらも2になっているグループを捜し出せば良いので，Casesで指定するパターンは2がひとつ増え，DropとMapで削ぎ落とす余計な部分は最初の2つに変わります．結果は，「3と5と7」の組しか見付かりませんでしたが，Tableの20を増やしてもこれしか見付かりません．実は，三つ子素数はこの一組しか存在しないのです．不思議ですね．

```
Map[Drop[#, 2] &, Cases[差と素数のトリオ, {2, 2, ___}]]
```

```
{{3, 5, 7}}
```

同じように，四つ子素数と呼ばれる「n，n+2，n+6，n+8がすべて素数であるような数の組」も見付けることが出来ます．これまでと異なるのは，赤字の部分の差を隣り合った三つのペアそれぞれで計算していること，四つ子なので素数のカルテットになっていることです．

```
差と素数のカルテット =
 Table[{Prime[n + 3] - Prime[n + 2],
   Prime[n + 2] - Prime[n + 1], Prime[n + 1] - Prime[n],
   Prime[n], Prime[n + 1], Prime[n + 2], Prime[n + 3]},
  {n, 1, 20}]
```

```
{{2, 2, 1, 2, 3, 5, 7}, {4, 2, 2, 3, 5, 7, 11},
 {2, 4, 2, 5, 7, 11, 13}, {4, 2, 4, 7, 11, 13, 17},
 {2, 4, 2, 11, 13, 17, 19}, {4, 2, 4, 13, 17, 19, 23},
 {6, 4, 2, 17, 19, 23, 29}, {2, 6, 4, 19, 23, 29, 31},
 {6, 2, 6, 23, 29, 31, 37}, {4, 6, 2, 29, 31, 37, 41},
 {2, 4, 6, 31, 37, 41, 43}, {4, 2, 4, 37, 41, 43, 47},
 {6, 4, 2, 41, 43, 47, 53}, {6, 6, 4, 43, 47, 53, 59},
 {2, 6, 6, 47, 53, 59, 61}, {6, 2, 6, 53, 59, 61, 67},
 {4, 6, 2, 59, 61, 67, 71}, {2, 4, 6, 61, 67, 71, 73},
 {6, 2, 4, 67, 71, 73, 79}, {4, 6, 2, 71, 73, 79, 83}}
```

この個々のリストの中から，最初の3つの差が2,4,2になっているグループを捜し出せば良いので，Casesで指定するパターンは「{2,4,2,___}」となり，DropとMapで削ぎ落とす余計な部分は最初の3つに変わります．

```
Map[Drop[#, 3] &,
 Cases[差と素数のカルテット, {2, 4, 2, ___}]]
```

```
{{5, 7, 11, 13}, {11, 13, 17, 19}}
```

【BlankNullSequence（___）】*Mathematica*ではパターンマッチが非常に重要な役割を担っています．今回のように特定の形にマッチするものを抜き出したり，以前の例のように特定のパターンの文字列だけを検出することは，多くの実際的な作業において頻出します．その中でもアンダースコアを複数組み合わせたパターンは重要で，いわゆるワイルドカード，何にでもマッチするジョーカーです．詳細はまだ説明しませんが，アンダースコアの数は重要な意味を持つので，入力間違いには注意してください．