Python と SageMath

佐々木格　(信州大学理学部)

2022 年 7 月 28 日

概要

Python は非常に良くデザインされたプログラミング言語で，覚えやすく可読性の高いコードが書けることが特徴です。本講義の後半では数式処理システム SageMath（セイジ，以下 Sage と略）を学習します。

Sage は 100 個ほどの数学ソフトウェアを統合した大規模なソフトウェアで，基礎代数，微分・積分，整数論，暗号理論，数値計算，可換代数，群論，組み合わせ論，グラフ理論等の計算を行うことができます。手軽にグラフを描画することもできるし，数学の研究で本格的に使うこともあります。

Python には系 2 と系 3 の二つの系統があり，それらには完全な互換性はありません。Sage のプログラムは Python の文法で記述しますので，本講義では，まずは Python の基本事項を学び，後半で Sage を使った数学的な計算を紹介します。最新の Sage のプログラムは Python3 の文法でかきます。以下では，まず Python3 について解説を行います[*1]。

Python や Sage はフリーソフトウェアですから，インターネットから無料でダウンロードして自分のパソコンにインストールして使うことができます。これらは Windows，Mac，Linux 版がそれぞれ開発されており，大学の環境だけでなく，自分が普段使用しているマシンにインストールして自由に使うことができます。

[*1] 2019 年度までは Python2 を教えていたので，再履修の学生は注意してください。

第 I 部

Python の基礎

このプリントは Linux での実行を想定して書かれています。Windows で行う場合では次のような読み替えを行ってください。

- 端末 (terminal) → コマンドプロンプト，Windows PowerShell
- テキストエディタ → メモ帳[*2]
- ディレクトリ → フォルダ
- 記号「\(バックスラッシュ)」 → 記号「¥」

1　Python プログラムの実行手順

このプリントでは Python プログラムを実行する方法として次の 2 つを紹介します。

(1) Python のプログラムが書かれたファイルを作成して端末から実行する
(2) インタラクティブシェルを使う

1.1　準備

まず，デスクトップに情報処理 I のファイルを入れるフォルダを作ります。デスクトップで右クリックし「新しいフォルダを作成」を選んで，`dataproc1` というフォルダを作ってください。フォルダを作ったらそこに，好みのテキストエディタで `test.txt` というファイルを作成しましょう。

1.2　最初のプログラム (Hello World)

(1) の手順を詳しく紹介します。プログラムの実行の流れは次の図の通りです：

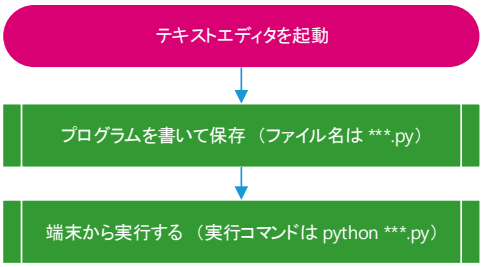


図 1　Python プログラムの実行方法 (1)

Python プログラムはテキストエディタで書きます。まず，デスクトップにある情報処理 I のフォルダ (`dataproc1`) をダブルクリックして開きますすると作成した `test.txt` という名前のファイルがあると思うので，これをテキストエディタで開きます。まずは，左上の「ファイル」をクリックし，「名前を付けて保存」を選びましょう[*3]。ファイル名を `hello.py` にして保存をします。これで，デスクトップにあるディレクトリ

[*2] もしくは好みのテキストエディタ (Visual Studio Code, Atom, Emacs, Sublime Text など)。Word は不可。
[*3] エディタによって場所は異なるけど，普通は同じ場所にあります。

dataproc1 にファイル hello.py が作られました。さて，hello.py を次の内容に書き直してください：

- ファイル名：**hello.py**

```
print('Hello World!')
```

入力したら保存します。これで最初のプログラムは完成です。このプログラムを実行するために，dataproc1 のフォルダに戻りましょう。daraproc1 のフォルダ上の空きスペースで [Shift]+[右クリック] し，端末 (PowerShell) を起動します。端末から次をタイプすることで Python プログラムが実行されます。

```
mint@mint-vb:~\Desktop\dataproc1$ python3 hello.py
Hello World!
mint@mint-vb:~\Desktop\dataproc1$
```

上のように端末に Hello World!と表示されれば成功です。

1.3 日本語を含む Python プログラム

Python2 では日本語を含む Python プログラムを書くには，文字コードを指定する必要がありましたが，Python3 からは不要になりました。次のプログラムを作成して，端末から python hellojp.py と実行してみましょう：

- ファイル名：**hellojp.py**

```
print('こんにちは')
```

このファイルの実行結果は次のようになるはずです。

```
mint@mint-vb:~\Desktop\dataproc1$ python3 hellojp.py
こ ん に ち は
mint@mint-vb:~\Desktop\dataproc1$
```

1.3.1 文字コードに注意

コンピューター内部ではデータは 0 と 1 の並びで表されています。その 0 と 1 の並びをアルファベットや漢字・記号等に対応させることで，人がテキストファイルを読めるようになります。その対応のルールを文字コードといいます。文字コードにはシフト JIS や UTF-8 といったものがあります。ある文字コードで書かれたファイルを，異なる文字コードで表示させると（当然ですが）正しく表示されず，いわゆる文字化けが起こるので注意しなければなりません。

Windows のメモ帳では 表示 (V) →ステータスバー (S) をクリックすると，右下に現在の文字コードを表示させることが出来ます。また，ファイルの保存画面でも文字コードを指定することができるので，確認してみてください。

シフト JIS は日本語専用の文字コードで昔は Windows の標準でしたが，現在は Windows でも UTF-8 が（ほぼ）標準になっています。Python がデフォルトで対応する文字コードは UTF-8 ですので，Python のプログラムはUTF-8で書いてください。それ以外の文字コードで書いた場合には，エラーがメッセージが表示されることがあります。

1.3.2 コメントアウト

Python ではプログラムのコメントは『#』の後に書きます[*4]。コメント部分は実行時には無視されます：

[*4] コメントの書き方は文章やプログラムによって異なります。例えば，LaTeX のコメントは % の後に書きます。

```
print('こんにちは')        # この部分は無視されます
```

このファイルの実行結果は hellojp.py と同じです。

1.3.3 全角スペースに要注意！！

さて，ここで最も重要な注意事項を説明します。日本語を含む文章やプログラムを書くときには

全角スペース「　←これ」に
常に注意しなければなりません。

スペースには半角スペース「 」と全角スペース「　」があり，全角スペースは半角スペース 2 個分のサイズですが，これらは**全く異なる**もので，しかも見た目で区別することが出来ません。

1.4 Python インタラクティブシェル

Python インタラクティブシェルは，入力したプログラムを順次実行していく簡易的なシステムです。インタラクティブシェルを起動するには端末から python [ENTER] と実行するだけです。

```
mint@mint-vb:~\Desktop\dataproc1$ python3
Python 3.8.10 (default, Mar 15 2022, 12:22:08)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

カーソルが最後の行で点滅して入力を待っています。四則演算と冪を試してみましょう：

```
>>> 1+3
4
>>> 5-3
2
>>> 4*3
12
>>> 9/4
2.25
>>> 10/3
3.3333333333333335    # 値は厳密ではない!
>>> 10/5
2.0          # 少数点数
>>> 10//5
2            # 整数
>>> 18//5
3            # 商を返す
>>> 12%5     # 割り算のあまり
2
>>> 2**10    # 2の10乗
1024
```

バックスラッシュ (/) は割り算の記号ですが，上のように厳密ではない値になることがあるので注意しましょう。また Python2 では 9/4) は商 2 を返すので Python2 を使うときには注意してください。多くのプログラミング言語では冪は 2^10 のように表しますが，Python では冪は 2^{10} は 2**10 のように表します。後半で解説する Sage のユーザーでは^は冪を意味するように変更されています。また 5/3 は $\frac{5}{3}$ という有理数（厳密）を表すように変更されています。

全角スペースの実験をしてみましょう。

```
>>> print('あ　い')      # 「あ」と「い」の間には半角スペースが存在する
あ　い
>>> print('う　え')      # 「う」と「え」の間には全角スペースが存在する
う　え
>>> print('お　　か')    # 「お」と「か」の間には半角スペースが2個存在する
お　　か
```

Python では文字列以外の場所に全角スペースを用いるとたちまちエラーとなります。次では和の計算の途中に全角スペースを入れてしまいました。

```
>>> 4+6        # 4+6を計算
10
>> 4 +6        # 4と+の間に半角スペースがある
10             # けどこれはOK
>>> 4  +6     # 4と+の間に全角スペースがある
  File "<stdin>", line 1
  4  +6
   ^
SyntaxError: invalid character in identifier
>>>
```

初心者はエラーメッセージに SyntaxError:　invalid character と表示されたら全角スペースの存在を疑ってください。そして，全角スペースは極力使わないようにしましょう。

最後に，インタラクティブシェルを終了するには exit() と入力します：

```
>>> exit()                          # もしくは ctrl+d（Linux or Macの場合）
mint@mint-vb:~\Desktop\dataproc1$
```

2　変数，予約語，文字列，数値，データの型

様々なデータを扱う上で最も基本となる事項を紹介します。

2.1　変数

変数を使った計算を，インタラクティブシェルを用いて紹介します。

```
>>> a = 6          # 変数aに6を代入
>>> a              # aの内容を表示
6
>>> a = 8          # 変数aの値を8に変更
>>> a
8
>>> a = a + 5      #aに5を足す
```

```
>>> a
13
>>> a += 1          #aに1を足す
>>> a
14
```

上のプログラムで a=a+5 は a の値を a+5 に変える事を意味しています。このように多くのプログラミング言語では = は恒等式ではなく代入を意味します。

さて，存在しない文字を呼ぶとどうなるでしょうか？

```
>>> b
Traceback (most recent call last):                    # エラーメッセージ
  File "<stdin>", line 1, in <module>                 # エラーメッセージ
NameError: name 'b' is not defined                    # エラーメッセージ
>>> b = -4                                            # bに-4を入れる
>>> a+b
10
```

定義されていない変数を使おうとすると上のようにエラーメッセージが出ます。変数名は一文字である必要はありません。

```
>>> ame = 4
>>> mikan = 5
>>> ame + mikan
9
```

変数名はある程度自由に決めることができますが，次のような決まりがあります。変数名はアルファベット a–z, A–Z, から始めなければならず，大文字と小文字が区別されます。先頭以外では，数字 0–9, やアンダーバー『_』を使うことが出来ます。それと次に紹介する『予約語』を変数名としては使うことはできません。

2.2 予約語

次の単語は Python の文法上，特別な意味を持つので，変数名として使ってはいけません：

Python の予約語

```
print and for if elif else del is raise assert import from
lambda return break global not try class except or while
continue exec pass yield def finally in
```

2.3 文字列

ここまでに変数，文字列，数値を扱いました。数値は 65, −3, 9.23 等と表されたもの，文字列は'Hello' のようにコーテーションマークで囲まれたもの，変数は文字列や数値等のデータを名前を付けて管理するためのものです。それぞれについてもう少し詳しく解説します。

2.3.1 文字列の定義

文字列はコーテーションマーク' 'や" "で囲まれたものとして定義されます。

```
>>> x = 'hello world'
>>> x
```

```
'hello world'
```

ここで x は変数で，'hello world' が文字列です。2 つの文字列は + でつなげることができます。

```
>>> aa = 'Alice'     # 変数aaにAliceを代入
>>> bb = ' and Bob'
>>> cc = aa + bb
'Alice and Bob'        # 文字列aaとbbがつながっている
>>> print(cc)
Alice and Bob
```

最後のように文字列を print するとコーテーションマークがとれたものが表示されます。

2.3.2 文字列の中でのコーテーションと改行

文字列は『" "』か『' '』で囲んで定義しますが，文字列の中でコーテーションマークを使いたい場合にはこれらを使い分けます。

```
>>> a = "This is a 'pen'."
>>> print(a)
This is a 'pen'.
```

改行を含む文字列は次のように『\n』を挿入して作ります：

```
>>> a = 'aaa\nbbb\nccc'
>>> a
'aaa\nbbb\nccc'
>>> print(a)       #printすると\nの部分は改行される。
aaa
bbb
ccc
```

『\n』を使わずに改行をするには，""" """または''' ''' で囲みます。

```
>>> a = '''aaa
... bbb
... ccc'''
>>> a
'aaa\nbbb\nccc'
>>> print(a)
aaa
bbb
ccc
```

また『\』を使って，改行文字や記号『\』，『"』，『'』などを表すことが出来ます。代表的な例を紹介しておきます：

\改行	⇝	改行を無視する
\\	⇝	\
\"	⇝	"
\'	⇝	'
\n	⇝	行送り

例えば

```
>>> aa = 'コーテーションマークとは\"などのこと'
>>> print(aa)
コーテーションマークとは"などのこと
```

2.4 データの型

文字列は str 型 (string) と呼ばれます。

```
>>> type('hello')        # type('hello')の型を調べる
<class 'str'>             # str型である
```

数値の型でよく使うものに整数型 (int) と浮動小数点数 (float) があります。

```
>>> type(123)       # 123の型は？
<class 'int'>       # 整数型
>>> type(3.14)      # 3.14の型は？
<class 'float'>     # float型
>>> a = 3.14        # 変数a に3.14を代入する
>>> type(a)
<class 'float'>     # 変数aの表すデータ(3.14)はfloat型
```

上では int は整数型 (integer)，float は浮動小数点数 (floating point number) を意味しています。整数型の演算は厳密に行われますが，浮動小数点数の計算では誤差が生じます。どちらも数であることは同じのですが，Python の内部での処理の仕方が異なるために，異なるデータの型として認識されるわけです。では，int 型と float 型の和は何になるでしょうか？

```
>>> aa = 10             # int型
>>> bb = 3.14           # float型
>>> cc = aa + bb
>>> cc
13.14
>>> type(cc)
<type 'float'>          # float型
```

答えは float 型でした。扱う数値がすべて整数の場合に（除法を行わなければ）常に厳密な値で計算できますが，float が一つでも混ざる場合は，厳密性が失われてしまうので注意が必要です。数値計算の際には，自分が扱う数値がどの型なのかを，常に認識しなければなりません。

2.5 数値と文字列の変換

さて文字列と数値は足せるでしょうか？

```
>>> 'Alice' + 1999
Traceback (most recent call last):                              # エラーメッセージ
  File "<stdin>", line 1, in <module>                           # エラーメッセージ
TypeError: can only concatenate 'str'(not "int") to str         # エラーメッセージ
```

このように str 型と int 型を足すことはできません。文字列と数値をくっつけるには，数値を文字列に変換する必要があります。

文字列，整数，小数の変換

str()：数値を文字列に変換する（例：str(123)='123'）
int()：文字列の数字を，整数に変換する（例：int('123')=123）
float()：文字列の数字を，浮動小数点数に変換する（例：float('123.45')=123.45）

次のプログラムはうまくいくはずです：

```
>>> aaa = str(1999)        # 数値1999を文字列にしたものをaaaに入れる
>>> aaa
'1999'
>>> type(aaa)
<class 'str'>              # aaaの型は確かにstringである
>>> 'Alice' + aaa          # 文字列'Alice'とaaaをつなげる
'Alice1999'
```

逆に，「文字列になっている数字の列」から整数や小数点数に変換してみましょう：

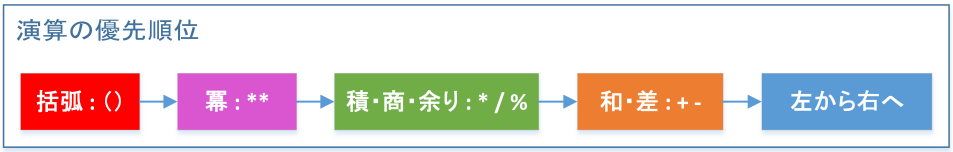
```
>>> aa = '123'             # '123'は文字列
>>> int(aa)                # aaを整数にして返す
123
>>> float(aa)              # aaを浮動小数点数にして返す
123.0
```

2.6　演算子と計算の順序

Python では，数値計算は次の演算子 (operator) で行います：

記号	意味	例
+	和	10+20 は 30 を与える
-	差	20-10 は 10 を与える
*	積	4*5 は 20 を与える
/	割り算	10/5 は float 型の数 2.0 を返す。
//	商	10/5 は Int 型の数 2 を返す。
%	余り	8%5 は余り 3 を与える
**	冪	2**3 は $2^3 = 8$ を与える

括弧で囲んだ部分は，例外なく最優先で計算されます。演算の優先順位は次のようになっています：



例えば，5*6/2**2 は，内部では次のような順で計算されているわけです：

```
5*6/2**2 = 5*6/(2**2)          # 冪が最初に計算される
         = 5*6/4
         = (5*6)/4             # 左から計算される
         = 30/4
         = 7.5
```

ただし，上の順序が適用されない例外がただ一つだけあります。それは冪の計算です。たとえば：

```
>>> 3**3**3
7625597484987
>>> (3**3)**3
19683
>>> 3**(3**3)      # 3**3**3はこちらに等しい
7625597484987
```

この例では，冪は右から順に計算されています。

プログラムには自分で気づかないうちにエラーが混入してしまうものです。単純なミスを防ぐためにも，積や除法を含む計算では，括弧で囲んで順番を明確にするのがよいでしょう。

3　プリント (print)

インタラクティブシェルでは「返されたもの」が順次画面に表示されますが，ファイルから Python を実行する場合は print しない限り，画面には何も表示されません。次のファイルを作り端末から実行してみましょう。

- ファイル名：**print01.py**

```
a = 6
print(a)      # これと
b = 4
a + b
print(a+b)    # これが表示される
```

実行結果の例

```
6
10
```

結果を見ればわかるように，3 行目，4 行目については何も出力がなく，2 行目, 5 行目で print された数字だけが出力されています。また print 文が複数回あるときは，改行されて表示されます。print 文の後に改行をさせたくない場合は『, end=" "』を書きます：

- ファイル名：**print02.py**

```
print(6, end=" ")  # 改行されないで一つスペースが入る
print(4)
```

実行結果の例

```
6 4
```

4　練習問題

4.1　問題：$e^{\pi} - 20$ の計算

pp = 3.141592，ee = 2.718281 とする。$\texttt{ee}^{\texttt{pp}} - 20$ を計算して表示 (print) する Python のプログラムを作成せよ。ファイル名は problem01.py とし，端末から

```
mint@mint-vb:~\Desktop\dataproc1$ python3 problem01.py
```

のように実行したときに，その数値を表示するようなプログラムでなければならない。

ちなみに，$e^{\pi} - 20$ は円周率に非常に近い値をとるが，これには何か理由があるのか，それとも偶然なのかわかっていない。

5　注意点

5.1　セミコロン;

次のようなプログラム

```
>>> a = 5
>>> b = 3
>>> c = a + b
>>> c
8
```

は，各文をセミコロンで区切って

```
>>> a = 5; b = 3; c = a+b; c
8
```

のように書くこともできます。セミコロンを多用するとプログラムの流れが見づらくなるので，あまり使わないほうがよいかもしれません。

5.2　代入演算子

Python では等号 = は，右辺のデータを左辺の変数に入れるときに用いますそこで記号 = は代入演算子 (assignment operator) と呼ばれます。式 3=4 は，数学では偽の命題ですが，Python では意味がない文なのでエラーとなります。

```
>>> 3=4
  File "<stdin>", line 1
SyntaxError: cannot assign to literal
```

変数 a の値に 5 を加えたい場合は

```
>>> a = 3
>>> a = a + 5    # aの値を5増やす
>>> a
8
```

のようにします。2 行目の a=a+5 に違和感を感じるかもしれませんが，これは = が代入を意味することを考えればおかしくはありません。次を試してみましょう。

```
>>> a = 3
>>> b = a     # bはaの値である。
>>> b
3             # もちろんbの値は3
>>> a = 4     # aの値を変える
>>> b
3             # bの値は変わらない
```

ここでも，b=a によって「a の値」が b に代入されるわけですが，変数 b が変数 a を参照しているのではないことに注意しましょう。

変数を箱にたとえてみます。a=3 によって箱 a に数値 3 が入ります。b=a とすると，箱 b に a の中身 3が入ります。箱 a の中身を 4 に変えても箱 b の中身は変わりません。

前にも少し説明しましたが，変数の数値を増やしたり減らしたりする複合代入演算子 +=，-=があります。

```
>>> a = 4
>>> a += 1
>>> a
5
>>> a -= 2  # a の値を 2 減らす
3
```

5.3 インタラクティブシェルと print

インタラクティブシェルでは変数の中身を知りたければ，上のように変数名 +Enter と入力すればよかったのですが，ファイルから実行する場合には，変数名を書いただけでは何も表示されません。例えば，次のプログラム（ファイルから実行する）では何も表示されません。

- ファイル名：**print03.py**

```
a = 3
a
```

ファイルから実行する場合に a の値を表示したければ，print を使います。

- ファイル名：**print04.py**

```
a = 3
print(a)
```

実行結果の例

```
3
```

6 様々なデータの形式

6.1 リスト (list)

リストとはいくつかのデータ (要素) の集まりです。まずはリストを作る事から始めましょう。

```
>>> a = ['Alice', 'Bob', 2,3,8]    # リストを定義して変数 a に入れる
>>> a                              # a の内容を確認
['Alice', 'Bob', 2,3,8]
```

当然ですが，リストの型は list です。

```
>>> type(a)                        # a の型を確認
<class 'list'>                     # a の型はリスト
```

続いて，リストの成分を呼び出すには次のようにします：

```
>>> a[0]                           # a の第 1 成分
'Alice'
>>> a[-1]                          # a の最後の成分
8
>>> a[3]                           # a の 4 番めの成分
3
>>> a[-2]                          # a の最後から 2 番めの成分
3
```

上のようにリストの成分の番号は 0 から始まります。存在しない成分を呼び出すとエラーになります：

```
>>> a[8]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

スライスという機能を使って，リストの一部を切り出すことができます：

リストからの要素の取り出し（スライス）

- a[:n] は最初の n 個の要素からなる新しいリスト
- a[-n:] は最後の n 個の要素からなる新しいリスト
- a[n:] は最初の n 個の要素を取り除いた新しいリスト
- a[:-n] は最後の n 個の要素を取り除いた新しいリスト
- a[n:m] は最初の n 個と最後の m 個を除いた新しいリスト

具体的なリストを作ってスライスしてみましょう：

```
>>> a = ['a','b','c','d','e','f','g','h']
>>> a[:3]
['a', 'b', 'c']
>>> a[-4:]
['e', 'f', 'g', 'h']
```

次にリストを足す (+) と結合されたリストが返されます：

```
>>> a = [1,2,3]
>>> b = ['a', 'b', 'c']
>>> c = a + b
>>> c
[1, 2, 3, 'a', 'b', 'c']
```

リスト自身への要素の追加は append() という『メソッド』を使って行います：

```
>>> a.append('Alice')
>>> a
[1,2,3,'Alice']
```

変数の値を代入で書き換えるように，リストの値も代入で書き換えることが出来ます：

```
>>> a = [1,2,3]
>>> a[0] = 'Alice'      #a[0]をAliceにする
>>> a
['Alice', 2, 3]
```

次に，リストから要素を削除するには pop(), remove() を使います：

```
>>> aa = ['a', 'b', 'c', 'd', 'e', 'b', 23, 8, 13]
>>> aa.pop()    #aaの末尾の要素を取り除く
13
>>> aa
 ['a', 'b', 'c', 'd', 'e', 'b', 23, 8, 12]     # 最後の13が無くなった
>>> aa.pop(3)     #aaから4番目の要素を取り除く
'd'               #4番目の要素'd'が除去された
>>> aa
['a', 'b', 'c', 'e', 'b', 23, 8]    #'d'が無くなっている
>>> aa.remove('b')      #aaにある最初の'b'を取り除く
```

```
>>> aa
['a', 'c', 'e', 'b', 23, 8]      # 二つ目の'b'は削除されない
```

連続する数字列からなるリストは次のようにして作ることもできます。

```
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> list(range(3,6))
[3, 4, 5]
>>> list(range(-2,4))
[-2, -1, 0, 1, 2, 3]
```

6.2 タプル (tuple)

タプルとはプログラミングや数学でしか聞かない言葉ですが，これは組または順序を持つ組を意味します。タプルもリスト (list) のように要素を集めたものですが，リストと異なるところは，変更が出来ないことです。タプルは要素を丸括弧（）で囲ってコンマで区切ります：

```
>>> a = (3,7, 'abc')      # タプル(tuple)を定義
>>> a
(3, 7, 'abc')
>>> type(a)
<class 'tuple'>      # aの型はタプル
>>> a.pop()          # aの最後の要素を除くことはできるか
raceback (most recent call last):
  File "<stdin>", line 1, in <module> # エラーメッセージ
AttributeError: 'tuple' object has no attribute 'pop'
```

タプルの要素を変更しようとするとエラーが起きます。タプルのように変更不可能なものを Immutable(イミュータブル) といいます。文字列，数値，タプルは Immutable なデータです。変更してはいけない大事なデータを誤って変えないために，このようなデータの型が用意されています。

6.3 辞書 (dictionary)

キー (key) と値 (value) の対応を集めたものを辞書といいます。辞書は次のようにして作ります：

```
>>> a = {'birth':1534, 'type':'A', 'name':'Nobunaga', 'death':1582}
```

これは織田信長のプロフィールです。辞書のキーを指定すると対応する値を呼び出すことが出来ます：

```
>>> a['birth']
1534
```

`keys()` や `values()` を用いることでキーのリストと値のリストを得られます：

```
>>> a.keys()        # aの『鍵』の集まりを返す
['death', 'type', 'name', 'birth']
>>> a.values()      # aの『値』の集まりを返す
[1582, 'A', 'Nobunaga', 1534]
```

辞書から要素を削除するには，`del` を使います。削除したいキーを指定すれば，値も同時に削除されます：

```
>>> del a['type']      # 鍵'type'とその値を削除
>>> print(a)
{'death': 1582, 'name': 'Nobunaga', 'birth': 1534}
```

辞書に要素を追加したい場合は，新しいキーと対応する値を次のように指示します：

```
>>> a['hobby'] = 'tea'
>>> print(a)
{'hobby': 'tea', 'death': 1582, 'name': 'Nobunaga', 'birth': 1534}
```

辞書のキーは immutable でなければなりません。つまりリストはキーにはなれませんが，タプルをキーにすることは出来ます：

```
>>> a = {(1,1,0):'police', (1,1,9):'fire', (1,7,7):'weather'}
>>> print(a)
{(1, 1, 0): 'police', (1, 1, 9): 'fire', (1, 7, 7): 'weather'}
```

キーと値をペアにしたタプルからなるリストをつかって，辞書を定義することも出来ます：

```
>>> a = [(1,'One'), (2,'Two'), (3,'Three')]
>>> b = dict(a)
>>> print(b)
{1: 'One', 2: 'Two', 3: 'Three'}
```

6.4 集合 (set)

要素を集めたものにリストやタプルがありましたが, 順番を気にしない要素の集まりが set です。immutable なものだけが set の要素になることができます。集合は次のように定義します：

```
>>> a = {1,4,3,2,2,2}
>>> a
{1,2,3,4}     # 2の重複したが無くなり，順番が変化している。
>>> a.add(5)       # aに5を付け加える
>>> print(a)
{1, 2, 3, 4, 5}
```

要素はソートされ，重複が除かれているのがわかります。関数 `set()` を使い，リストを集合に変換することもできます。

```
>>> a = [1,2,3]
>>> b = set(a); b
{1,2,3}
```

集合の演算 $\cup, \cap, \setminus$ などの演算も用意されています。

- ファイル名：**set01.py**

```
A = {1,2,3,4}
B = {3,4,5,6}
print(A | B)    # A∪B
print(A & B)    # A∩B
print(A - B)    # A\B
```

実行結果の例

```
{1,2,3,4,5,6}
{3,4}
{1,2}
```

集合そのものに加えたり，共通部分を取ったりする複合代入演算子

`|=`, `&=`, `-=`

が用意されています。例えば，

```
>>> a = {1,2,3}; b = {3,4}
>>> a |= b    # 集合aに集合bの要素を加える。
>>> a
{1,2,3,4}
```

となります。

7　キーボードからのデータの取得

キーボードから入力した文字や数値に応じて，プログラムの結果を変化させる事を考えます。キー入力を取得するには input() という関数を使います。次のファイルを作って実行してみましょう：

- ファイル名：**input1.py**

```
namae = input('あなたの名前を入力してください。：')
print('こんにちは', namae, 'さん')
```

上で作ったファイルを実行すると，次のような表示が出ます：

```
mint@mint-vb:~\Desktop\dataproc1$ python3 input1.py
あなたの名前を入力してください。：
```

そこで，信州花子と入力して Enter キーを押せば

```
こんにちは 信州花子 さん
mint@mint-vb :~\Desktop\dataproc1$
```

と出力されます。上のプログラムは次の手順を実行しました：

1. 「あなたの名前を入力してください。：」と画面に表示
2. 変数 namae を作成。キーボード入力を待つ。
3. 入力された文字列を変数 namae に代入
4. 「こんにちは・・・さん」と画面に表示

input() で入力された文字列は変数 namae に格納されるわけです。

実は，入力するものが数値である場合には上の手順では不十分です。Python の input() 関数は，入力されたデータを常に文字列として扱うからです。数値にするには，int 関数か float 関数を使って，文字列としての数を適切な数値データに変換する必要があります。次はキー入力を受け付けて，入力した数値の 2 乗を出力するプログラムです。

- ファイル名：**input2.py**

```
aa = input('数値（整数）を入力してください：')    # aaは入力された文字列になる。
num = int(aa)          # ここで文字列を整数(int型)に変換
print('入力された数値の2乗は', num**2, 'です。')
```

これを実行すると次のようになります：

```
$ python3 input2.py
数値(整数)を入力してください：9
入力された数値の2乗は 9 です。
$
```

上のプログラムで，入力を 1.5 などの整数以外にするとエラーとなります。少数点数を扱いたい場合は，入力された文字列を float 関数を使って浮動小数点数に変換します。

上の二つを合わせて次のような BMI を計算するプログラムを作ってみましょう：

- ファイル名：**bmi1.py**

```
1 namae = input('あなたの名前を入力してください：')
2 shintyo = input('あなたの身長は何センチですか？：')
3 shintyo = float(shintyo)  # shintyoを浮動小数点数に変換
4 weight  = input('あなたの体重は何キログラムですか？：')
5 weight = float(weight)    # weightを浮動小数点数に変換
6 bmi = 10000*weight/(shintyo**2)
7 print(namae, 'さんのBMIは', bmi, 'です。')
```

プログラムを実行して自分の BMI を計算してみましょう。

8　論理型と比較演算子

数学における，正しい主張は真，そうでないものは偽であるといいます。Python では真を True，偽を False で表します。これらは予約語であり特別な意味を持ちます。文法に注意しながら，インタラクティブシェルでの例を見てください：

```
1 >>> a = 3      #aに3を代入
2 >>> a == 3    #aは3だろうか？
3 True           #a==3は正しい！
4 >>> a == 2    #aは2だろうか？
5 False          #a==2は正しくない
6 >>> a > 2
7 True
8 >>> 3 != 2    #3は2に等しくないだろうか？
9 True           #3と2は異なるので，上の条件は真
```

この例のように，二つの数値を比較して真か偽かの条件を調べる事ができます。上の例で記号『==，>，!=』を使いました。これらはデータを比較するときに用いるので**比較演算子**と呼ばれています。数値に対して使える使える比較演算子には次があります：

数値に対して使える比較演算子

==　!=　<　>　<=　>=

これらは，それぞれ数学記号 $=$，$\neq$，$<$，$>$，$\leq$，$\geq$ に対応しています。

不等号と等号の順番は英語の”less than or equal” の順番と同じだと憶えましょう。『!=』は”not equal” と憶えます。

True と False には論理演算 and，or，not を行うことが出来ます。

```
1 >>> a = True       # aをTrueとする
2 >>> b = False      # bをFalseとする
3 >>> a and b       # aかつbは？
4 False
5 >>> a or b        # aまたはbは？
6 True
7 >>> not a         # aの否定は？
8 False
```

不等号は次のように連続して使うことができます。

```
>>> a = 4
>>> 3 < a < 5
True
```

上の 2 行目は 3<a and a<5 と同じ意味の文です。

リストや集合などコレクションに対して使える比較演算子もあります：

リストや集合などに対して使える比較演算子

==　　!=　　in

==と!= の説明は不要でしょう。in は次のように用います：

```
>>> a = [3,6,5,1]      #aをリスト[3,6,5,1]とする
>>> 5 in a             #5はaの中にいるだろうか？
True
>>> 7 in a             #7はaの中にいるだろうか？
False
```

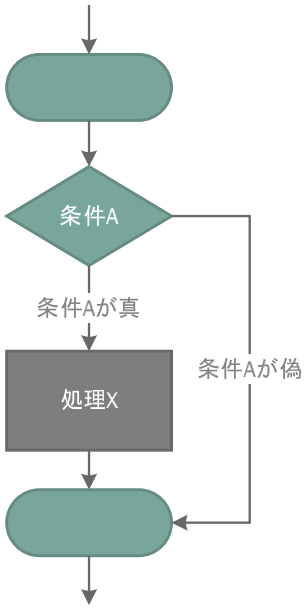
次のようにして，型を調べる type と組み合わせて使うことも出来ます：

```
>>> a = [3,5,6,1]
>>> type(a)
<class 'list'>
>>> type(a) == list
True
>>> type(a) == tuple
False
```

9　条件分岐

9.1　if 文

条件 (True か False) に応じて，処理を変化させるときに `if` 文という決められた文法を用います。if 文の手順は次のチャートの通りです：



この手順を Python では次のように書きます：

Python での if 文の構造

```
if  条件A:
    [   処理X   ]
    [   処理X   ]
    [   処理X   ]
```

- 条件 A が `True` であれば処理 X が行われ，`False` であれば処理 X は行われません。
- `if` の行の行末のコロン『：』を忘れずに！
- 処理 X は，上のように揃えた字下げをおこなう。

ここで，処理 X の前のインデント（字下げ）が文法上重要な役割を果たします。処理 X が数行にわたるときに，どこまでが処理 X であるかはインデントによって判断されます。インデントは単なるスペースなのですが，Python がプログラムを実行するときにインデントが揃った部分が，`if` 文で処理されるプログラムの塊（ブロック）であると判断します：

if 文を使った例題をやってみましょう。次の簡単なアルゴリズムを Python プログラムでかきます。

(1) 整数 a を入力させる。
(2) a が偶数なら，『a is even』とプリントしてから，a を半分にする。
(3) a が奇数なら何もしない。

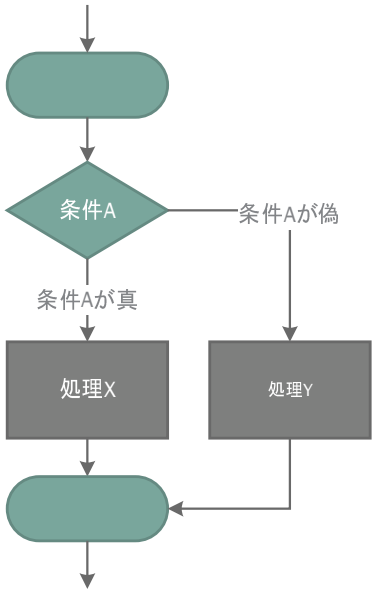
- ファイル名：**if1.py**

```
1 a = input('Input integer a: ')
2 a = int(a)  # aを整数にする
3
4 if a%2 == 0:       # もしaが偶数なら
5     print('a is even')   # a is evenと表示
6     a = a//2             # そして，aを半分にする。インデントに注意。
7
8 print(a)           # aの値を表示
```

上のプログラムの 4〜6 行目が if 文です。上のプログラムを実際に実行していくつかの数値を入れてみて，想定通りになるか試してみましょう。

9.2 if-else 文

次に，少し複雑な条件分岐を考えます。条件が真のときには処理 X，偽のときには別の処理 Y がしたいとしましょう。



もちろん，これは上で説明した if 文だけで書くことができます。つまり

```
1 if 条件A:       # 条件Aが真なら
2     処理X       # 処理Xを行い，偽なら行わない。
3 if not 条件A:   # 条件Aが偽なら
4     処理Y       # Yを行い，そうでなければ行わない。
```

と書くだけです。しかし，次に説明する if-else 文を使って，よりわかりやすく書くこともできます。

Python での if-else 文の構造

```
if 条件A:
    処理X
else:
    処理Y
```

- 条件 A が `True` であれば，処理 X を行い，`False` であれば処理 Y を行う。
- ここでも処理 X，処理 Y はインデントによって判断されます。

`if-else` 文を使った例題をやってみましょう。次のアルゴリズムを考えます。

(1) 整数 `a` を入力させる。
(2) もし `a` が偶数ならば，`a` を半分にする。
(3) そうでなければ，`a` を `3*a+1` にする
(4) `a` の値を表示する。

これをプログラムで書いてみましょう。

- ファイル名：**if2.py**

```
a = input('a?: ')
a = int(a)          # 入力したものを整数に変換する。

if a%2 == 0:        # もしaが偶数なら
    a = a//2        # aを半分にする
else:               # そうでなければ
    a = 3*a + 1     # aを3a+1にする

print(a)            # aの値をプリント
```

プログラムを実行していくつかの動作を試してみましょう。

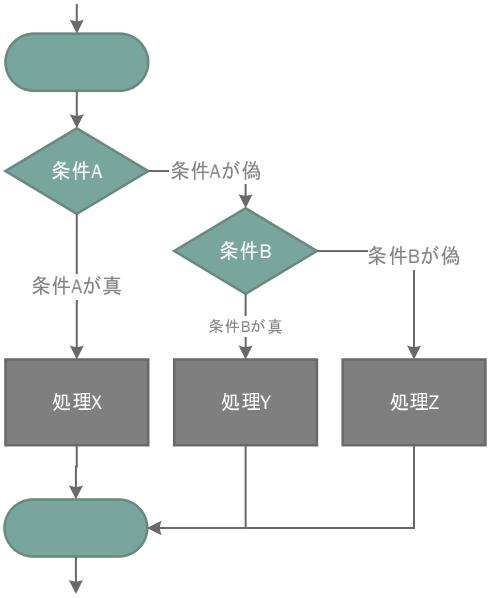
9.3 if-elif-else 文

さらに条件分岐を記述するためには if-elif-else 文を使います。elif は else if の略です。

if-elif-else 文の構造

```
if 条件A:
    処理X
elif 条件B:
    処理Y
else:
    処理Z
```

- 条件 A が真の場合に処理 X を行います。
- 条件 A が偽で条件 B が真のときに，処理 Y を行います。
- 条件 A も条件 B も成り立たないときに，処理 Z を行います。
- さらに条件 C，条件 D，・・・と条件が続く場合は `elif` で処理します。



さらに `elif` を追加して，`if-elif-elif-elif-else` のように条件を増やすことができます。

9.4　複雑な if 文の例

`if-elif-else` 文を使って，与えた西暦が閏年かどうかを判定するプログラムを作りたい。グレゴリウス暦では閏年は次のように決められている：

- 西暦年が 4 で割り切れる年は閏年とする。
- ただし西暦年が 4 で割り切れる年でも，100 で割り切れる年は閏年としない。
- ただし西暦年が 4 で割り切れ，100 でも割り切れる年でも 400 で割り切れる年は閏年とする。

上の閏年の定め方の文章には，『ただし』による条件が後からついていて，そのままではプログラムにはしづらい。そこで，まず閏年の条件を次のようにわかりやすい同値な条件に直します：

1. 西暦年が 400 で割り切れるならば，それは閏年である。
2. 上以外のとき，西暦年は 100 で割り切れるならば，それは閏年ではない。
3. 上以外のとき，西暦年が 4 で割り切れるならば，それは閏年である。
4. 上のどれにも当てはまらないとき，その年は閏年ではない。

これを実現する Python のプログラムは次のようになります。

- ファイル名：**uruuQ.py**

```
year = input('西暦を入力:')    # 入力した数値を変数yearに代入する
year = int(year)

if year % 400 == 0:      # もしyearを400で割った余りが0なら
    print(year, 'は閏年です。')
elif year % 100 == 0:    # そうでないとき，もしyearが100で割り切れたら
    print(year, 'は閏年ではありません。')
elif year % 4 == 0:      # そうでないとき，もしyearが4で割り切れたら
    print(year, 'は閏年です。')
else:                    # 上の全ての条件に当てはまらないとき
    print(year, 'は閏年ではありません。')
```

10　練習問題

10.1　問題：BMI 計算プログラム

名前，身長，体重を入力させ，そこから BMI を計算・表示し，その値に応じてやせ・肥満度の結果を出すプログラムを作りたい。プログラムは次の手順を行うものとする。

(1) 名前を入力させ，変数名 namae に代入
(2) 身長・体重を入力させ，それぞれ shintyo, weight に代入
(3) bmi を表示
(4) もし bmi<18.5 なら，「やせ気味です」と表示
(5) そうでないとき，bmi<25.0 なら，「ふつうです」と表示
(6) そうでないとき，bmi<30 なら，「太り気味です」と表示
(7) うえのどちらでもないとき「太りすぎです」と表示

次のプログラムの『*****』の部分を推測して，上の手順が実行されるようにプログラムを完成させなさい。

- ファイル名：**bmi2.py**

```
namae = input('あなたの名前を入力してください：')
shintyo = input('あなたの身長は何センチですか？：')
shintyo = ***(shintyo)
weight  = input('あなたの体重は何キログラムですか？：')
weight = ***(weight)

bmi = 10000.0*weight/(shintyo**2)
bmi = round(bmi,1)     #bmiの値を小数点以下2桁目を四捨五入する

print(namae, 'さんのBMIは', bmi, 'で', end='')

if **********:
    print('やせ気味です。')
elif **********:
    print('ふつうです。')
**** **********:
    *****('太り気味です。')
****:
    *****('太りすぎです。')
```

11　For 文による繰り返し

11.1　For 文の文法

いくつかの処理の繰り返しを記述する方法を解説します。たとえば，hello をプリントするという処理を 5 回繰り返すには手動で

```
>>> print('hello')
hello
>>> print('hello')
hello
>>> print('hello')
hello
>>> print('hello')
hello
>>> print('hello')
hello
```

とやればよいのですが，次に説明する for 文を使えば，これを自動的に行うことができます：

```
for j in range(5):
    print('hello')
```

ここで，list(range(5)) はリスト [0,1,2,3,4] を返すことを思い出しましょう。実際に，Python のインタラクティブシェルでこのプログラムを実行してみます：

```
>>> for j in range(5):
...     print('hello')
...
hello
hello
hello
hello
hello
```

上の for 文は j が 0 から 4 まで，その下の処理を繰り返すので，次のように j の値を使うこともできます：

```
for j in range(5):
    print(j)
```

実行結果の例

```
0
1
2
3
4
```

ここで j はダミー変数で，他の文字に変えても結果は同じです。例えば，次は上のものと同じです：

```
for x in range(5):
    print(x)
```

さて，for 文の構造は次のようになっています：

For 文の構造

同じ作業を 10 回繰り返すプログラムです：

```
1 for j in range(10):
2     |こ こ の ブ ロ ッ ク に|
3     |繰 り 返 す          |
4     |プ ロ グ ラ ム を 書 く|
5
6 <次 に 行 う プ ロ グ ラ ム は こ こ に 書 く>
```

- 2〜4 行目のインデント（字下げ）されている部分が繰り返す処理。
- 6 行目は，インデントから外れるので for 文の外にあるので繰り返されない。この部分は for 文の繰り返しが終わってから実行される。

図 2　for 文の構造

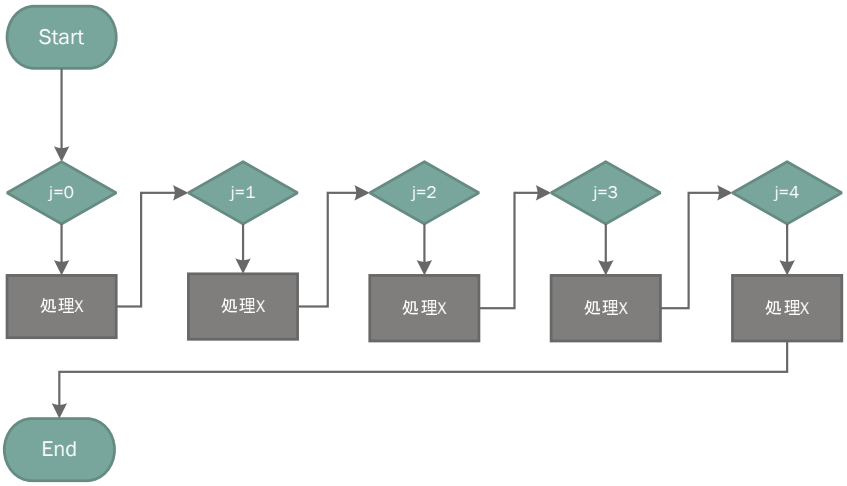


図 3　for 文で行われる処理 (5 回の繰り返し)

11.2　for 文の簡単な例

for 文のプログラムをいくつか書いて実行結果を見てみましょう：

- ファイル名：**for1.py**

```
1 for j in range(5):              # 以 下 の ブ ロ ッ ク を 5 回 繰 り 返 す
2     print('wanwan', end=' ')    # こ れ と
3     print('nyannyan')           # こ れ を 繰 り 返 す
```

実行結果の例

```
wanwan nyannyan
wanwan nyannyan
wanwan nyannyan
wanwan nyannyan
wanwan nyannyan
```

次のプログラムの最後の行は for 文のインデントから外れるので一回しか実行されません：

- ファイル名：**for2.py**

```
1 for j in range(5):                # 以下のブロックを5回繰り返す
2     print('wanwan', end=' ')      # この行と
3     print('nyannyan')             # この行を繰り返すが
4
5 print('gaogao')                   # ここは繰り返さない
```

実行結果の例

```
wanwan nyannyan
wanwan nyannyan
wanwan nyannyan
wanwan nyannyan
wanwan nyannyan
gaogao
```

for 文を使って 0 から 9 までの数字の 2 乗を表示します：

- ファイル名：**for3.py**

```
1 for j in range(10):                # jを0から9まで変えて次を繰り返す
2     print(j, j**2)                 # jとjの二乗をプリントする
3
4 print('owari')                     # ここは繰り返さない
```

実行結果の例

```
0 0
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
owari
```

数字を文字列にしてつなげれば次のような操作もすぐにできます：

- ファイル名：**for4.py**

```
1 for j in range(1,101):
2     print('ひつじが', j , '匹')
```

実行結果の例

```
ひつじが 1 匹
ひつじが 2 匹
ひつじが 3 匹
. . . . .
. . . . .
. . . . .
ひつじが 99 匹
ひつじが 100 匹
```

与えられたリストに対して，その要素を順に取り出して処理をすることができます：

- ファイル名：**for5.py**

```
1 aa = ['Alice', 'falls', 'down', 'a', 'rabbit', 'hole.']      # リストaaを定義
2
3 for i in aa:          # aaの中のiについて順番に
4     print(i, end=' ')          # iをプリントする
```

実行結果の例

```
Alice falls down a rabbit hole.
```

11.3 for 文でありがちなエラー

Python の文法の特徴にインデントで構文を判断するというものがありました。`if-else` 文ではインデントを間違えるとエラーになりましたが，`for` 文でも同じ事がおきます。次の例のように字下げが少し違うとエラーとなります：

- ファイル名：**for6.py**

```
1 for j in range(5):
2     print('wanwan')
3   print('nyannyan')
```

実行結果の例

```
  File "for6.py", line 3
    print('nyannyan')
                    ^
IndentationError: unindent does not match any outer indentation level
```

次のようにインデントをさげた場合も同じくエラーとなります：

- ファイル名：**for7.py**

```
1 for j in range(5):
2     print('wanwan')
3      print('nyannyan')
```

実行結果の例

```
  File "for7.py", line 3
    print('nyannyan')
                    ^
IndentationError: unindent does not match any outer indentation level
```

11.4 for 文の応用 1 ：for 文の中に for 文を入れる

もちろん for 文の中に for 文を入れて二重に繰り返す事もできます。

- ファイル名：**for8.py**

```
1 for i in range(5):              # i = 0 ~ 5に対して以下を繰り返す
2     for j in ['a','b','c']:     # j = a, b, cに対して次を繰り返す
3         print(i, j)             # iとjをプリント
```

実行結果の例

```
0 a
0 b
0 c
1 a
...
...
4 c
```

次のようにすれば掛け算の表が出力されます：

- ファイル名：**for9.py**

```
1 for i in range(1,10):             # i=1,2,...,9と
2     for j in range(1,10):         # j=1,2,...,9に対して
3         print(i*j, end=' ')       # i*jをプリント
4     print('')                     # ここで改行
```

実行結果の例

```
1 2 3 4 5 6 7 8 9
2 4 6 8 10 12 14 16 18
3 6 9 12 15 18 21 24 27
4 8 12 16 20 24 28 32 36
5 10 15 20 25 30 35 40 45
6 12 18 24 30 36 42 48 54
7 14 21 28 35 42 49 56 63
8 16 24 32 40 48 56 64 72
9 18 27 36 45 54 63 72 81
```

11.5　for 文の応用 2：if 文と組合わせる

if 文と組合わせて，for 文の中で処理を分岐させるとより複雑なプログラムが書けます。次では j が偶数なら j is even，奇数なら j is odd と表示させるようなプログラムです：

- ファイル名：**for10.py**

```
1 for j in range(1,10):
2     if j%2 == 0:
3         print(j, 'is even')
4     else:
5         print(j, 'is odd')
```

実行結果の例

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
```

11.6　for 文の応用 3：break で処理を止める

for 文では与えられた回数だけ繰り返すことが簡単にできますが，処理を途中で止めたいときには break を使います。for 文の繰り返し中に break が現れたら for 文は直ちに終了し，次の処理に進みます。例えば次のようなプログラムを考えます。

- 2^{2^j} で j を $1, 2, 3, \cdots, 10$ と動かして次々プリントする。
- しかし，この数列はものすごい早さで増加するので，その値が 100000000 を超えた場合には，その時点で for 文処理を停止したい。

- ファイル名：**for11.py**

```
1 for j in range(11):
2     a = 2**2**j
3     if a<1000000000000:
4         print(a)
5     else:
6         print('Too big!!')
7         break    # これが実行されたらfor文はおしまい
```

実行結果の例

```
2
4
16
256
65536
4294967296
Too big!!
```

11.7　for 文の応用 4：差分

直線状を動く物体の 0.1 秒ごとの位置のデータがリストとして

```
xlis = [0.39, 0.64, 0.84, 0.96, 1.0, 0.95, 0.81, 0.6, 0.33, 0.04]
```

と与えられているとしよう。このとき，開始時刻を 0 とし，0.1 秒ごとの平均の速度を表示するには次のようにすればよい：

- ファイル名：**for12.py**

```
xlis = [0.39, 0.64, 0.84, 0.96, 1.0, 0.95, 0.81, 0.6, 0.33, 0.04]
dt = 0.1                          # dt=0.1[秒]とする。
for j in range( len(xlis)-1 ):    # 繰り返しの回数は(データの個数)-1回
    v = (xlis[j+1] - xlis[j])/dt        # 時刻jからj+1の平均の速度
    print(v)
```

実行結果の例

```
2.5
1.9999999999999996
1.2
0.40000000000000036
-0.5000000000000004
-1.399999999999999
-2.1000000000000005
-2.6999999999999993
-2.9000000000000004
```

さて，実際のデータ解析の場面では，はじめから位置データのすべてが xlis のようには与えられていない場合が多い。例えば，リアルタイムに GPS の位置情報を取得し速度を算出するといった場合には，各時刻 (または過去) の位置情報は持っているが，未来の位置情報は手元にはない。他には，データーが巨大なファイルで与えられており，巨大ファイルを一度にメモリに読み込むことができない，といった場合がある。

このような場合を想定し，次のプログラムのように各時刻のデータを扱うことを考えよう：

```
xlis = [0.39, 0.64, 0.84, 0.96, 1.0, 0.95, 0.81, 0.6, 0.33, 0.04]

for x in xlis:
    *********
    *********
```

ここで言いたいことは，このゲームの制限は 1,3 行目にような形でのみ位置情報 xlis が使えるということである。このとき，for12.py のように各時刻の平均の速度を表示するプログラムを書くにはどのようにしたらよいだろうか？

この場合，for 文では各時刻の位置が一度だけ使えるので，ひとつ前の時刻の位置情報は x 以外の変数に保存しておく必要がある。これを解決するためには，時刻 −0.1 秒の位置を仮に y=0 とセットし，位置情報 x が読み込まれたとき，0.1 秒前の位置が y になるようにすればよい。手順を具体的に書くと次のようになる：

1. dt=0.1 とする。
2. y=0 とする。
3. 0 秒の位置 x=0.39 を得る。
4. −0.1 から 0 秒の平均の速度 (x-y)/dt=3.9 をプリント。
5. y に x=0.39 を入れる。
6. 0.1 秒の位置 x=0.64 を得る。
7. 0 秒から 0.1 秒の平均の速度 (x-y)/dt=2.5 をプリント。
8. y に x=0.64 を入れる。

9. ··· 繰り返し ···

この手順を実行するプログラムは次のようになる：

- ファイル名：**for13.py**

```
xlis = [0.39, 0.64, 0.84, 0.96, 1.0, 0.95, 0.81, 0.6, 0.33, 0.04]
dt = 0.1
y = 0
for x in xlis:       # xlisの位置情報を順に読み出しxとおく
    print((x-y)/dt)
    y=x
```

実行結果の例

```
3.9                  ←これだけ違う事に注意!
2.5
1.9999999999999996
1.2
0.40000000000000036
-0.5000000000000004
-1.399999999999999
-2.1000000000000005
-2.6999999999999993
-2.9000000000000004
```

この結果と for12.py の結果を比べると，最初の速度だけが異なる[*5]が，それ以外は同じ結果となったことがわかるだろう。

さて，速度の差を時間間隔で割ったものが平均の加速度である。上と同じ位置情報 xlis について，平均の加速度を返すプログラムは次のようになる。

- ファイル名：**for14.py**

```
xlis = [0.39, 0.64, 0.84, 0.96, 1.0, 0.95, 0.81, 0.6, 0.33, 0.04]
dt = 0.1
for j in range( len(xlis)-2 ):    # 繰り返しの回数は(データの個数)-2回
  a = ((xlis[j+2]-xlis[j+1]) - (xlis[j+1]-xlis[j]))/(dt*dt) # 平均の加速度
  print(a)
```

実行結果の例

```
-5.0000000000000036          ← 0 秒から 0.2 秒の間の平均の加速度
-7.999999999999995           ← 0.1 秒から 0.3 秒の間の平均の加速度
-7.999999999999995
-9.000000000000007
-8.999999999999984
-7.000000000000016
-5.999999999999988
-2.000000000000007
```

となる。

12　ファイルの書き込み・読み込み

Python の実行結果をファイルに書き込んで保存する方法を解説します。

12.1　ファイルの書き込み（端末の機能を使う）

ファイルの書き込みで手軽なのは端末の機能を使うことです。Python プログラムでファイルに書き出したいものを print しておき，端末から

```
***/dataproc1$ python3 ファイル名.py > out.txt
```

[*5] 最初の 0.3900000000 は時刻 −0.1 秒の位置を仮に 0 としたことから出てきた無意味なものなのでデータを利用する際は無視すればよい。

のようにすることで，本来 print されたものが out.txt に書き込まれます。ここでファイル out.txt がなければ自動的に作られます。例えば，for3.py の結果を for3result.txt に書き出すには

```
***/dataproc1$ python3 for3.py > for3result.txt
```

とします。ファイルの末尾に『追記』するには，『>>』を使います。試しに

```
***/dataproc1$ python3 for4.py >> for3result.txt
```

を実行してみましょう。これで，先程作ったファイル for3result.txt の末尾から，for4.py の実行結果が追記されました。ファイルを開いて確認してみましょう。

12.2　ファイルの書き込み（Python の機能を使う）

ファイルに書き込むもう一つの方法は Python 自体のファイル操作の機能を使うことです。

- ファイル名：**write1.py**

```
abc = 'Taro Hanako'             # 変数abcを定義
f = open('test.txt', 'a')       # 追記モード(a)で開く
f.write(abc)                    # abcをtext.txtに書き込む(追加)
f.close()                       # ファイルを閉じる
```

端末から

```
***/dataproc1$ python3 write1.py
```

と実行すると，新たにファイル test.txt が作られて Taro Hanako という文字列が書き込まれました。もう一度実行すると，同じファイルに Taro Hanako が追加されます。追記ではなく，すでにあるものを消去し上書きするには上の 3 行目の代わりに

```
f = open('test.txt', 'w')       # 書き込みモード(w)で開く
```

とします。

12.3　ファイルの読み込み

ファイルを読み込むには read や readlines といった命令を使います。read はファイルの全部の内容を一つの文字列として読み込み，readlines はファイルの各行をリスト化して扱います。

ここで，読み込むファイルを作るために次を実行します：

```
***/dataproc1$ python for10.py    # for10.pyの結果の確認
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
***/dataproc1$ python for10.py > guuki.txt    # 結果の出力
```

for10.py の実行結果が guuki.txt に書き出されました[*6]。次は，このファイルを使った read の例です。

- ファイル名：**read1.py**

```
f = open('guuki.txt', 'r') # 読み込みモードで開く
aaa = f.read()             # guuki.txtの内容をaaaとする
print(aaa)                 # aaaの内容をプリントする
f.close()                  # guuki.txtを閉じる
```

実行結果の例

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
```

つぎに，各行を readlines で読み込んで見ましょう：

- ファイル名：**read2.py**

```
f = open('guuki.txt', 'r')  # 読み込みモードで開く
aaa =  f.readlines()        # guuki.txtの各行からなるリストをaaaとする
print(aaa)                  # aaaの内容をプリントする
f.close()                   # guuki.txtを閉じる
```

実行結果の例

```
['1 is odd\n', '2 is even\n', '3 is odd\n', '4 is even\n', '5 is odd\n', '6 is even\n', '7 is odd\n', '8 is
```

aaa は guuki.txt の各行が文字列になったリストである事がわかります。改行は「\n」になっています。

13　練習問題

13.1　問題：$3n+1$ 問題

$3n+1$ 問題というものがあります。自然数 n に対して

- もし n が偶数なら n を半分にする
- もし n が奇数なら n を $3n+1$ にする

という操作を繰り返すと，最終的にはどんな自然数も 1 になるであろうという予想です。現在も未解決の問題で角谷の問題とか Collatz 予想と呼ばれています。例えば，最初の数が 9 の場合上の手順で作られる数列は

9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

となります。入力された数 a に対して，上の手順で作られる数を次々に表示するプログラムを作りたい。プログラムは次のアルゴリズムに従うように作りたい。

1. 自然数 a の値を入力させる。
2. 最大の繰り返し回数 maxiter を 1000 とする。
3. 以下の 4 から 7 を for 文で maxiter 回繰り返す。
4. もし a の値が 1 なら break で for 文を終了する

[*6] Windows PowerShell では guuki.txt の文字コードが UTF-16 で保存されるようです。このまま python で読み込むと文字化けするので，一度メモ帳などで guuki.txt を開いて文字コードを UTF-8 にして保存してください。

5. もし a が偶数なら a を半分にする
6. もし a が奇数なら a を $3a+1$ にする
7. a をプリントする

以下の Python プログラムの****を自分で考えて上のアルゴリズムが実現するようにしなさい。

- ファイル名：**collatz1.py**

```
a = int(input('a?: '))
print(a)

maxiter = 1000
for i in range(maxiter):
    *********:
        break
    **************:
        a = a//2
    ****:
        *********
    print(a)
```

プログラムが完成したら実行して a として 13 を入力してみましょう。プログラムが正しければ，出力は次のようになるはずです。

実行結果の例

```
a?: 13
13
40
20
10
5
16
8
4
2
1
```

（注意）上のプログラムでは最大の繰り返し回数を 1000 に設定しましたが，何回繰り返すのかがやってみないとわからない処理を書くには `while` 文を使います (1000 回では足りないかもしれない！)。`while` 文は次の章で学習します。

13.2 順次与えられた位置情報から加速度を算出する問題

`for13.py` と同様に，位置情報が `for x in xlis:`のように得られた場合に，平均の加速度を表示するプログラムを作りたい。プログラムは，$-0.2, -0.1$ 秒の位置を仮に `x0=0`，`x1=0` とおいて，次の手順を行うものとする。

1. `dt = 0.1` とする。
2. `x0 = 0` とする。
3. `x1 = 0` とする。
4. (`for` 文で) 位置情報 `x` を得る。
5. 初めの 0.1 秒間の平均の速度を `v0=(x1-x0)/dt` とする。
6. 次の 0.1 秒間の平均の速度を `v1=(x-x1)/dt` とする。
7. 最初の 0.2 秒間の平均の加速度を `a = (v1-v0)/dt` とする。
8. `a` の値をプリント。
9. 変数 `x0,x1` に `x1,x` の値を入れる。
10. 手順 4 から 9 を繰り返す。

この手順を実行するプログラムを次のように書く。

- ファイル名：**for15.py**

```
xlis = [0.39, 0.64, 0.84, 0.96, 1.0, 0.95, 0.81, 0.6, 0.33, 0.04]
dt = 0.1   # dtは時間間隔
x0 = 0     # 変数を
x1 = 0     # 2つ用意する
for x in xlis:    # 位置情報xを順次読み込む
    **********
    **********
    **********
    **********
    **********
```

ただし，最初に出力される 2 つのデータは無意味なものである。

上の for15.py を完成させましょう。ただし，****の部分ではリスト xlis を使ってはいけません。この問題がわからない人は for13.py が理解できていないので，for13.py の解説を見直し，これがどのような動作をしているのかもう一度確認してください。正解の場合 for15.py の出力は次のようになるはずです。

実行結果の例

```
39.0
-13.999999999999998
-5.000000000000004
-7.999999999999996
-7.999999999999996
-9.000000000000007
-8.999999999999986
-7.000000000000015
-5.999999999999988
-2.0000000000000107
```

14　while 文による繰り返し

上の 13.1 節の問題は $3n+1$ 問題に関するもので，与えられた自然数 n に対して，もし n が偶数なら n を半分にし，n が奇数なら n を $3n+1$ にする，という操作を n の値を表示しながら，最終的に $n=1$ になるまで繰り返し行うものでした。そこでは，for 文を使い，繰り返しの回数は最大 1000 回までと決めていましたが，この数列は何回のステップで $n=1$ となるか分からないし，そもそもどんな自然数から始めても最後に $n=1$ になることは証明されていません（反例も見つかっていません）。このように，何回繰り返すかわからない場合は for 文ではなく，次に説明する while 文によって記述するのが適当です。

14.1　while 文の文法

ある処理の繰り返し行うときに，繰り返す回数がわかっていれば for 文を使いますが，**繰り返す回数が分からない場合や永遠に繰り返しを行う場合には while 文によってプログラムを記述します。**

while 文

```
while 条件A:
    [  処理X  ]
    [  処理X  ]
```

- 条件 A が `True` であるあいだ，ずっと処理 X を繰り返す。
- 条件 A が `False` なら処理 X は行われず while 文は終了する。もし A が `True` だったら処理 X が実行され，X の終了後にまた条件 A をチェックします。同様に，A が `True` ならば処理 X を行い `False` なら while 文は終わり。この繰り返しは，条件 A が `True` である限りずっと続きます。
- オプションとして `break`, `continue` が用意されています：
  - 処理 X の実行中に `break` が現れると強制的に while 文から抜ける。
  - 処理 X の実行中に `continue` が現れたら，処理 X を中断し条件 A を確認するプロセスに戻る。

図 4　while 文の文法

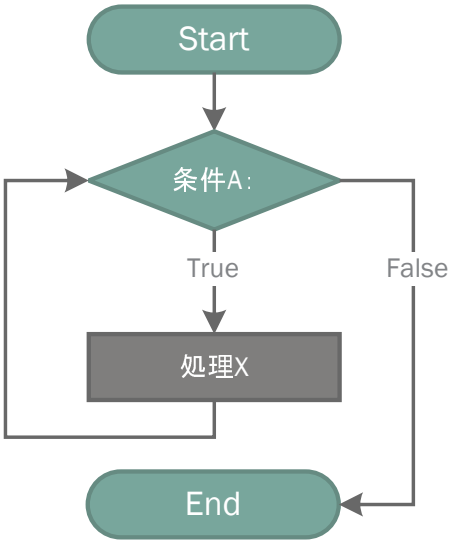


図 5　while 文の処理の流れ

次の while 文の簡単な例を見てみましょう。

- ファイル名：**while1.py**

```
a = 1
while a<10:    # aが10未満なら，3-4行目を行う。
    print(a*a, end=' ')     # aの二乗をプリントする
    a = a+1         # aの値を一つ増やす。2行目に戻る。

print('owari')    # while文を抜けたらこの処理を行う
```

実行結果の例

```
1 4 9 16 25 36 49 64 81 owari
```

ここで，上の説明での条件 A に相当するものは『a<10』で，処理 X は 3,4 行目です。

上のプログラムのように，while 文の中にカウンター（一つずつ増える変数）と停止条件を書くことで，for 文と同じ処理を行うことができます。上のプログラム while1.py と同じ事を for 文で書くこともできます。

```
for a in range(1,10):
    print(a*a, end=' ')

print('owari')
```

14.2　while 文の例

14.2.1　$3n+1$ 問題の数列の生成

while 文を用いて前の節の問題（n が偶数なら半分にし，奇数なら $3n+1$ に変えることを繰り返してできる数列を作る問題）を書き直してみましょう：

- ファイル名：**while2.py**

```
a = int(input('Input an integer: '))    # 数を入力させて，その数をaに入れる
print(a, end=' ')       # aの値をプリント

while a != 1:          # a≠1である限り以下の処理を繰り返し行う
    if a%2 == 0:       # aが偶数なら
        a = a//2
    else:              #aが偶数でないなら
        a = 3*a+1
    print(a, end=' ')
```

実行結果の例

```
Input an integer: 19
19 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

前回の課題と同じ結果が得られました。もっと大きな数ではどうなるでしょうか？例えば a=6171 から始めると 261 回の繰り返しの末に最終的に $a=1$ となります。

実行結果の例

```
Input an integer: 6171
6171 18514 9257 27772 13886 6943 20830 10415 31246 15623 46870 23435 70306 35153 105460
..........
... 略 ...
..........
1300 650 325 976 488 244 122 61 184 92 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1
```

前回のプログラム collatz1.py では繰り返しの回数に上限がありましたが，while2.py では無制限の繰り返しの計算を行うことができます（もちろんコンピューターの容量の制限の中でではありますが）。

14.2.2 繰り返しが終わらないプログラム

次に while 文を使って永遠に終わらないプログラムを作ってみましょう：

- ファイル名：**while3.py**

```
1 j=1
2 while True:      # 条件は永遠に真なので，以下をずっと繰り返す！
3     print('ひつじが'+str(j)+'匹')
4     j = j+1
```

実行結果の例

```
ひつじが 1 匹
ひつじが 2 匹
ひつじが 3 匹
  ...
  ...
```

この while 文では条件がいつでも真（True）なので繰り返し処理が永遠に続きます。プログラムを停止するには CTRL + C もしくは CTRL + Z を入力します。

14.2.3 while 文の中で break, else, continue を使った例

while 文の繰り返しは break を書くことによって止めることができます：

- ファイル名：**while4.py**

```
1 j=1
2 while True:
3     print('ひつじが'+str(j)+'匹')
4     j=j+1
5     if j > 123456:  # もし羊の数が123456を超えたらwhile文を終える
6         break
7
8 print("Too much sheeps! I'm already asleep...zzz")
```

実行結果の例

```
ひつじが 1 匹
ひつじが 2 匹
ひつじが 3 匹
...
...
ひつじが 123456 匹
Too much sheeps! I'm already asleep...zzz.
```

while 文でも else を使うことができます。while 文の条件が False になったときだけに行う処理を書きます。while 直後の条件が成立しなければ else 以下を処理します：

- ファイル名：**while_else.py**

```
1 a=0
2 while a<5:
3    a = a+1
4    print(a)
5 else:
6    print('owari')
7
8 print('END')
```

実行結果の例

```
1
2
3
4
5
owari
END
```

上のプログラムは意味のないプログラムですが，後で紹介するように break を while-else 文の中で使うと効果的なプログラムを作ることができます。

次に while 文の中で continue を使った文を作ります。その前に，ある文字列の中に特定の文字 (列) が入っているかどうかを確認する方法を紹介します。次は文字列 Shinshu の中に ns をいう文字が入っているかを調べています。

```
>>> 'ns' in 'Shinshu'
True
>>> 'sn' in 'Shinshu'
False
```

さて，羊を数えたいのですが，数字の 4 は不吉に感じるので，匹数に 4 を含むものは飛ばしながら最大 15 匹まで数えることにしましょう。

- ファイル名：**while_continue.py**

```
a=0
while a<15:
    a=a+1
    if '4' in str(a):    # もしaの中に数字4が含まれていたら
        continue         # 次のprintを行わずにwhileの条件確認に戻る
    else:                # そうでなければ
        print('羊が'+str(a)+'匹') # 羊を数える

print('zzz...')
```

実行結果の例

```
羊が 1 匹
羊が 2 匹
羊が 3 匹
羊が 5 匹
羊が 6 匹
羊が 7 匹
羊が 8 匹
羊が 9 匹
羊が 10 匹
羊が 11 匹
羊が 12 匹
羊が 13 匹
羊が 15 匹
zzz...
```

実は上のような処理は for 文を用いたほうが簡潔に書けます：

```
for i in range(1,16):
    if not '4' in str(i):    # もしiの中に数字4が含まれていないのなら
        print('羊が'+str(i)+'匹')

print('zzz...')
```

14.2.4 フィボナッチ数列の生成

フィボナッチ数列 $1, 1, 2, 3, 5, 8, 13, 21, 34, \cdots$，つまり漸化式

$$a_1 = 1, \quad a_2 = 1, \quad a_{n+1} = a_n + a_{n-1}, \quad n = 2, 3, 4, \cdots \tag{1}$$

で定義される数列を表示するプログラムを作ってみましょう。ただし，数列の値が 100000000 を超えたら停止するものとします。この場合でも，いつ a_n が 100000000 を超えるか分からないので while 文を使います。プログラムは次のアルゴリズムに従って書くことにしましょう：

1. maxvalue = 100000000 と置く。
2. a=1 と置く。b=1 と置く。
3. while 文で a<maxvalue が成り立っている間は，次の処理 4-5 を繰り返す。
4. a の値をプリントする。
5. a, b の値をそれぞれ b, a+b に置き換える。
6. a<maxvalue が偽になって while 文を抜けたら，次の数列の値をプリントする。

- ファイル名：**while5.py**

```
1 maxvalue = 100000000    # maxvalueを右辺の数字にセットする
2 a = 1      # a = 1 とおく
3 b = 1      # b = 1 とおく
4 while a < maxvalue:
5     print(a)            # ここが
6     a, b = b, a+b       # 繰り返されるブロック
7
8 print('The next value is', a)     # the next value isとaの値をプリントする
```

実行結果の例

```
1
1
2
3
5
...
63245986
The next value is 102334155
```

14.2.5 素数判定プログラム

次に与えられた自然数 (≥ 2) が素数かどうか判定するプログラムを作ってみましょう。与えられた数 n を 2 から順に $\sqrt{n}$ 以下のすべての自然数で割ってみて，どれかの数で割り切れたら n は合成数，そうでなければ n は素数です。そこで次の手順で判定する事にします：

1. 数 n の値を入力させる (ただし $n \geq 2$ とする)。
2. $i = 2$ とする。
3. $i^2 \leq n$ である間は，次の手順 4〜5 を繰り返す。
4. もし，n で i で割り切れたら「n は素数ではありません」とプリントして 3 の繰り返しを終了する。
5. そうでなければ i を $i + 1$ にする。
6. もし 3 の条件全てが偽であったら，「n は素数です」とプリントする。

- ファイル名：**while_primeQ.py**

```
1 n = input('Input an integer(>1): ') # キーボードから入力した数を変数nとする
2 n = int(n)     # nを整数に変換
```

```
3
4  i = 2                  # i=2 とおく
5  while i*i <= n:        # i*i≦nなら以下を繰り返す
6      if n % i == 0:     # もしnがiで割り切れたら
7          print(n, 'is not a prime.')  # nは素数ではないとプリント
8          break                        # while文抜けて12行目以下へ進む
9      else:              # そうでなければ
10         i += 1         # iの値を一つ増やす
11 else:                          # while文の条件が偽となったら
12     print(n, 'is a prime.')    # nは素数であるとプリントする。
13
14 print('おわり')
```

実行結果の例

```
Input an integer: 1237
1237 は素数です。
おわり
```

実は上のプログラムでは $n = 1$ が素数と判定されてしまいます。

14.3　プログラミング言語とはなにか

プログラミング言語とはコンピューターでの処理を記述するための仕組みです。C，Python，Java など様々なコンピューター言語がありますが，書きやすさや速度の問題を別にすれば，これらはすべて同等の能力を持ちます。例えば，C 言語でできる処理は Python でもできるし，その逆も成り立ちます。これはコンピューター言語がチューリング完全という性質を持つためです。大雑把に言えば，データを任意に読み書きでき，与えられた条件に応じて処理を変化させる事ができ，任意の回数繰り返しができるような言語はチューリング完全です。Python については，リストの操作と if 文，及び while 文だけで（メモリの有限性を除けば）チューリング完全になったといえます。これだけ知っておけば，原理的には，皆さんの工夫次第で，人工知能を作ったり，微分積分をさせたりといった処理が可能なはずなのです。

コンピューターを使うときに最初に必要となる心構えは，自分がすでに万能チューリングマシンを持っているという認識ではないかと思います。

15　練習問題

15.1　問題：素数判定プログラム

上のプログラム（`while_primeQ.py`）を修正し，$n = 1$ が素数でないと判定されるようにしなさい。ヒント：

- 1,2 行目は同じ
- もし $n = 1$ なら，n `is not a prime.` とプリントする。
- そうでなければ，上のプログラムの 4〜12 行目を実行する（適切にインデントを！）。
- `print('おわり')` はなくてもよい。

15.2　問題：ユークリッドの互除法

ユークリッドの互除法とは自然数 a, b の最大公約数を求める次のアルゴリズムの事です。

(i) $b = 0$ なら a が最大公約数である。

(ii) $b \neq 0$ なら a と b の最大公約数は b と r の最大公約数に等しい。ただし r は a を b で割った余り。

次の手順を行う Python プログラムを書きなさい：

1. 整数 a, b を入力させる。
2. $b \neq 0$ である間は，次の処理 3,4,5 を繰り返し行う（while 文を使う）。$b = 0$ ならステップ 6 へ。
3. a を b で割った余りを r と置く。
4. a に b を代入
5. b に r を代入
6. $b = 0$ になったらそのときの a が最大公約数なので a をプリントする。

次がプログラムの例です：

- ファイル名：**gcd.py**

```
a = int(input('Input an integer: '))
b = int(input('Input an integer: '))

while b != 0:
    *********
    *********
    *********

print('The greatest common divisor is', a)
```

上の*****の部分を考え，プログラムを完成させなさい。

16　関数

特定の処理を繰り返すときには for や while で繰り返せばよいのですが，その処理をいろんな場所で自由に呼び出して使いたいときには**関数**を使うのが便利です。関数はひとかたまりの処理に名前を付けて，必要なところで使えるようにしたものです。

Python ではじめから用意されている関数があり，そのようなものを組み込み関数といいます。例えば，すでに何回も使っている

print(), int(), float(), str()

などは組み込み関数です。他にはリストの総和を求める関数 sum() や，最大値，最小値を返す max(), min() があります：

```
>>> sum([1,2,3,4,5])      # リストの合計を返す関数
15
>>> max([1,2,3,4,5])      # リストの最大値を返す関数
5
```

図 6　プログラムを再利用しよう

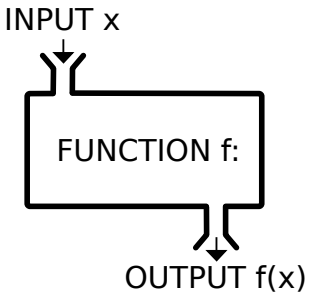


図 7　関数のイメージ

16.1　関数の定義のしかた

関数の定義

```
def 関数名(引数):                    # 引数は inputされるデータの変数名
    [処理 X を記述したブロック]       # 関数が実行するプログラム
    return [戻り値]                  # output
```

- 関数は def 文で定義します。
- def の後に関数名と引数を書いたら右端にはコロン『:』を付けます。
- 関数の定義が終わるまでインデントを保つ。
- 引数とは input されたデータを入れる変数名です。
- 引数や戻り値は不要なら省略できます。
- 関数名の大文字と小文字は区別されます。
- return は処理 X の中でも使える。return したら関数の処理は終わり。

それでは次の例題を見てみましょう。

- ファイル名：**func1.py**

```
def greeting():              # ここが
    print("I'm fine.")       # 関数の定義
```

```
3
4 print('How are you?')       # How are you?と表示する
5 greeting()                  # 関数を呼び出す
```

実行結果の例

```
How are you?
I'm fine.
```

上のプログラムでは greeting() は呼び出されたら print("I'm fine") を実行する関数です。関数は定義された段階 (上の例では 1,2 行目) では実行されず，プログラム中で呼びだされたときにだけ動きます。上の例では，関数は引数も戻り値も持ちません。

次に**引数と戻り値（return）**があるプログラムの例を見てみましょう。

- ファイル名：**func2.py**

```
1 def sanjou(a):        # 関数名はsanjou，引数はa
2    return a*a*a       # aの3乗を【返す(return)】
3
4 print(sanjou(3))      # sannou(3)で返される値をプリント
5 print(sanjou(10))     # sannou(10)で返される値をプリント
```

実行結果の例

```
27
1000
```

sanjou( ) は引数の三乗を返す関数ですが，上のように返された値をプリントしたり代入したりすることができます。次の例では引数 a が奇数なら odd，偶数なら even を返す関数を定義しています：

- ファイル名：**func3.py**

```
1 def guuki(a):
2   if a%2 == 0:          # もしaが偶数なら
3      return 'even'      # evenを返す
4   else:                 # そうでなければ
5      return 'odd'       # oddを返す
6
7 print(guuki(1232), guuki(99))
```

実行結果の例

```
even odd
```

次にもう少し実用的な関数を作ってみます。西暦 n 年に対してその年が閏年なら True を，そうでなければ False を返す関数 uruuQ を作ります。さらに西暦 1000 年から 2017 年までの閏年をすべて表示します。

- ファイル名：**func4.py**

```
1 def uruuQ(n):
2     if n%400 == 0:
3        return True
4     elif n%100 == 0:
5        return False
6     elif n%4 == 0:
7        return True
8     else:
9        return False
10
11 for i in range(1000,2018):
12    if uruuQ(i):        # もしi年が閏年なら
```

```
13        print(i, end=' ')        # iをプリントする
```

実行結果の例

```
1004 1008 1012 1016 1020 1024 1028 ..... 1984 1988 1992 1996 2000 2004 2008 2012
```

次に羊を好きなだけ数える関数を定義してみます：

- ファイル名：**func5.py**

```
1 def CountSheeps(a):
2     for i in range(1,a+1):              #iを1からaまで繰り返す
3         print('羊が' + str(i) + '匹')     # 羊がi匹
4
5 CountSheeps(45)       # 関数を呼び出す
```

実行結果の例

```
羊が 1 匹
羊が 2 匹
・・・
・・・
羊が 45 匹
```

16.2 関数とローカル変数

関数の定義の中で新しく定義された変数は，その定義の中だけで**一時的に利用**できます。このような変数のことを**ローカル変数**といいます。これは関数の処理をそこだけで完結させるために補助的に用いるものです。ローカル変数はそれが定義された関数の外では使えません。これによって変数に使う文字を節約することができます。ローカル変数でない変数を**グローバル変数**といいます。

次のような数学の問題を考えてみましょう。$N = 100$ とするとき，$S = \sum_{k=1}^{N} k^2$ を求めなさい。ここで，

$$S = 1 + 2^2 + 3^2 + \cdots + 100^2 = \sum_{k=1}^{N} k^2 = \sum_{j=1}^{N} j^2 = \sum_{\ell=1}^{N} \ell^2 \tag{2}$$

なので，Σ の和を取るための変数 k, j, ℓ に特に意味は無く，他の文字[*7]を使っても構いません。そして (2) の和で使われている k, j, ℓ は Σ の外では意味を持ちません。一方，S と N には決まった数が代入されています。S と N に対応するものが変数がグローバル変数で，k のようにある処理の外では意味が無い変数がローカル変数に対応します。

次のプログラムは台形の面積を返す関数を定義していて，c と s はローカル変数になります。

- ファイル名：**daikei.py**

```
1 def daikei(a,b,h):    #a上底，b下底，h高さ
2     c = a+b           #cはローカル変数
3     s = 1.0*c*h / 2   #sはローカル変数
4     return s
5
6 S = daikei(3,5,8)     # Sはグローバル変数
7 print(S)
8
9 print(c)    # これはエラー(cはここでは定義されていない)
```

[*7] ただし，他で使われていないもの

実行結果の例

```
32.0
Traceback (most recent call last):
  File "daikei.py", line 8, in ?
    print(c)
NameError: name 'c' is not defined
```

16.3 関数の説明の書き方

関数の定義を書くときに，その関数がどういう処理を行うのかをコメントとして書いておきましょう。コメントを書くことは次のような意味があります。

1. 関数の処理を書く前にコメントを書くことで，行いたい処理が明確になる。
2. 後になってプログラムを見た時に，何をやっていたのか思い出すことができる。。
3. 他人が読むときの助けになる。

Python では関数の定義（def）の直後にコメントを書く慣習があります。そこでは関数が行う処理をコーテーション『""" と """』で囲んで文字列として書きます。

たとえば，羊を数える関数 gcd(a,b) を定義するときには，次のようにコメントを書くとよいでしょう：

```
1 def CountSheeps(a):
2     """ 羊を a 匹数える関数 """
3     for i in range(1,a+1):              # iを1からaまで繰り返す
4         print('羊が' + str(i) + '匹')   # 羊がi匹とプリントする
5
6 CountSheeps(45)
```

上では 2 行目が関数の説明ですが，これは単なる文字列なのでプログラム実行時には何もしません。

16.4 def 文の応用：素数を判定する関数

以前のプリントで，与えられた数字に対してそれが素数かどうかを判定するプログラム (while_primeQ.py) を作りましたが，それをもとに素数判定の関数を定義してみましょう。次で定義する関数は，引数 n が素数なら True を返し，そうでなければ False を返します。

- ファイル名：**primeq.py**

```
1  def primeQ(n):
2      """ nが素数ならTrue, 合成数ならFalseを返す関数を定義 """
3      if n < 2:
4          return False
5      else:
6          i=2
7          while i**2 <= n:
8              if n%i == 0:    #nがiで割り切れるなら
9                  return False
10             i=i+1
11     return True
12
13 # 1から100までの素数を列挙する。
14 for k in range(1,101):
15     if primeQ(k):              # もしkが素数なら
16         print(k, end=', ')      # kをプリントする
```

実行結果の例

```
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97,
```

上のプログラムは小さい数に対してはすぐに答えを出しますが，例えば

$$111\ 111\ 111\ 111\ 111\ 111\ 111\ 113$$

のような大きい数が素数かどうか判定するには長い時間がかかってしまいます（実際この数は素数です）。大きな数の素数判定を行うにはそれなりの工夫が必要になります。多項式時間で素数を判定する AKS 素数判定法や確率的だが高速で素数判定を行う Miller-Rabin 素数判定法といったものが知られています。一般に高速に動作するプログラムを作成するには，高度な数学的知識が必要になります。後半で解説する Sage では素数かどうかを高速に判定する関数 `is_prime` がデフォルトで用意されています。

16.5 練習問題

16.5.1 最大公約数を計算する関数

15.2 節の練習問題を参考にして，与えられた 2 つの自然数 a, b の最大公約数を返す関数を定義しなさい。そして，その関数を使って 23954187074819 と 8326543 の最大公約数を求めなさい。次のファイルの*****を推測すればよい。

- ファイル名：**def_gcd.py**

```
1 def gcd(a,b):
2     """ aとbの最大公約数を求める関数
3         ユークリッドの互除法により最大公約数を計算する
4     """
5     ************:        # while文開始
6        **************    # (a,b)を(b,a%b)に同時に置き換える。
7     return *
8
9 print( gcd(75757, 94963) )
```

実行結果の例

```
1067
```

16.5.2 円周率を近似する分数

規約分数 a/b で円周率を近似するものを沢山見つけたい。初期条件を $a = 3, b = 1$ とし，$a/b < \pi$ なら a を増やし，$a/b > \pi$ なら b を一つ増やす，という操作を繰り返すことで，そのような既約分数を探してゆく。a/b が既約かどうかは，$\gcd(a, b) = 1$ かどうかで判断すればよい。

次のアルゴリズムを実行する Python プログラムを作成せよ。ファイル名は `approxPi.py` とすること。

(1) `pi=3.141592653589793` とする。
(2) `a` と `b` の最大公約数を返す関数 `gcd(a,b)` を定義する。
(3) 変数 `a`，`b`，`d` にそれぞれ `3`，`1`，`0.14` を入れる。(`d` は誤差評価に用いる)
(4) `a<32000` かつ `b<10000` である間，以下の (5)–(8) を繰り返す。
(5) `apr=a/b` とおく（円周率の近似値になる予定）。
(6) `err=abs(pi - apr)` とおく（ここに，`abs` は絶対値を返す組み込み関数）。
(7) もし `a` と `b` が互いに素かつ `err<d` ならば `a,b,apr,err` をプリントし，`err` の値を変数 `d` に入れる。
(8) もし `pi > apr` なら `a` の値を一つ増やし，そうでなければ `b` の値を一つ増やす。

16.5.3 ２つの自然数が規約である割合

自然数 m, n が 1 以外の公約数を持たないときに，それらは規約であるという。これはもちろん，m と n の最大公約数が 1 であることと同じである。$N = 10000$ とする。$1 \leq n, m \leq N$ について，(m, n) が規約であるものの個数 A とその割合 $A/10000^2$ を表示するプログラムを作成せよ。

補足：勝手に与えた自然数 m, n が規約である確率 P は

$$P = \left(\frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \cdots\right)^{-1} = \frac{6}{\pi^2} = 0.6079271018\cdots$$

であることが知られている（これは初等的な議論で示すことができる）。

16.5.4 フィボナッチ数列を返す関数

フィボナッチ数列は

$$a_1 = 1,\ a_2 = 1,\ a_n = a_{n-1} + a_{n-2},\quad n = 3, 4, 5, \cdots$$

で定義される数列である。具体的には $1, 1, 2, 3, 5, 8, 13, 21, 34, \cdots$ となる。a_n を計算するには次のようにすればよい，

(1) $n = 1$ または $n = 2$ なら 1 を返す。
(2) $a = 1, b = 1$ とする。
(3) 次の手順 (4) を $n - 2$ 回繰り返す。
(4) 変数 a, b の値を同時に $a + b, a$ に置き換える。
(5) 変数 a の値を返す。

上の手順で，フィボナッチ数列の第 n 項を計算する関数 `fib(n)` を定義し，`fib(1)`, `fib(2)`,$\cdots$, `fib(10)` を表示せよ。ファイル名は `fib0.py` とすること。

16.6 関数の再帰的な定義

階乗 $n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$ を返す関数を定義してみましょう。このような関数を定義するには for 文を使って，掛け算を繰り返せばよいです。

- ファイル名：**kaijou1.py**

```
def kaijou(n):
    """ nの階乗n!を返す関数をfor文を使って定義する """
    a=1
    for i in range(1,n+1):
        a = a*i
    return a

for i in range(1,10):       #i=1,...,9に対して
    print(kaijou(i), end=' ')       #kaijou(i)の値を表示する
```

実行結果の例

```
1 2 6 24 120 720 5040 40320 362880
```

さて，関数 $f(n)$ を漸化式で

$$f(0) = 1, \quad f(1) = 1, \quad f(n) = n \cdot f(n-1)$$

で定義すると明らかに自然数 n に対して $f(n) = n!$ となります。このような関数の定義の仕方を**再帰的な定義** (recursive definition) といいます。Python では関数を再帰的に定義することができます。次のプログラムは階乗 $n!$ を再帰的に定義したものです：

- ファイル名：**kaijou2.py**

```
def kaijou(n):
    """ n!を再帰的に定義する"""
    if n==0 or n==1:
        return 1
    else:
        return n*kaijou(n-1)        # ここで自分自身kaijou(n-1)呼び出す！

for i in range(10):                 # i=0,1,2,…,9に対して
    print(kaijou(i), end=' ')       # kaijou(i)をプリントする
```

実行結果の例

```
1 1 2 6 24 120 720 5040 40320 362880
```

最大公約数を求めるユークリッドの互除法も同じ手続きの繰り返しである事を利用して，gcd(a,b) を次のように再帰的に定義することも出来ます：

- ファイル名：**gcd_rec.py**

```
def gcd(a,b):
    if b == 0:                  # もしb=0なら
        return a                # aを返す
    else:                       # そうでなければ
        return gcd(b,a%b)       # gcd(b,a%b)を返す

print(gcd(1428,2618))
```

実行結果の例

```
238
```

上のプログラムでは，答えを得るまでに次のような動作が行われています：

1. a=1428，b=2618 として gcd(a,b) の中身を実行する。
2. $b = 0$ かどうかを調べたが，そうではないので gcd(2618,1428%2618) を返す。
3. 1428 ÷ 2618 の余りは 1428 なので gcd(2618,1428) を考える。
4. 1428 は 0 ではないので，gcd(1428,2618%1428) が返される。
5. 2618%1428=1190 なので gcd(1428,1190) を考える。
6. 1190 は 0 ではないので，gcd(1190,238) が返される。238=1428%1190。
7. 238 は 0 ではないので，gcd(238,1190%238) を返す。
8. 1190%238=0 と第 2 の変数が 0 になったので第 1 変数 238 を返す。
9. 返された値は print によって表示される。

上の例のように，関数がそれ自身による繰り返しで定義可能な場合は，簡潔に関数を定義することが出来ます。

16.7　再帰的な定義の落とし穴と計算量

上で見たように関数が再帰的に定義できるというのはとても便利ですが，実はステップごとに計算量が増えていくような関数の定義には不向きです。以下でこのことを見てみましょう。ここではコンビネーション ${}_mC_n$ を定義します。これは m 個の異なるものの中から，n 個取り出す場合の数です。高校で習ったように

$$ {}_mC_n = \frac{m!}{(m-n)!n!} \tag{3} $$

です。一方で，これは次の漸化式によって表すこともできます：

$$ {}_mC_n = \begin{cases} 0 & \text{if } \ m < n \text{ or } n < 0 \\ 1 & \text{if } \ m = 0 \text{ or } m = n \\ {}_{m-1}C_n + {}_{m-1}C_{n-1} & \text{それ以外} \end{cases} \tag{4} $$

ここで $n < 0$ のときや $m < n$ のときは ${}_mC_n$ は意味がないので 0 としました。

式 (3), (4) の両方を使って ${}_mC_n$ を計算してみましょう。次のプログラムでは (3) を使って定義する関数を C(m,n)，(4) を使う方を D(m,n) としています。

- ファイル名：**combinat1.py**

```
1  def kaijou(n):    # kaijou(n)=n!を定義
2      if n == 0 or n==1:
3          return 1
4      else:
5          return n*kaijou(n-1)
6
7  def C(m,n):     # (3)で組み合わせ数を定義
8      return kaijou(m)//(kaijou(m-n)*kaijou(n))
9
10 def D(m,n):     # (4)で組み合わせ数を再帰的に定義
11     if n<0 or m<n:
12         return 0
13     elif m == 0 or m==n:
14         return 1
```

```
15      else:
16          return D(m-1,n) + D(m-1,n-1)
17
18  for i in range(7):              # i=0,1,2,3,4,5,6に対して
19      print(C(6,i), end=' ')      # C(6,i)を表示
20
21  print('')    # 改行する
22
23  for i in range(7):              # i=0,1,2,3,4,5,6に対して
24      print(D(6,i), end=' ')      # C(6,i)を表示
```

実行結果の例

```
1 6 15 20 15 6 1
1 6 15 20 15 6 1
```

もちろん計算結果は同じです。でも少し大きな数になるだけで計算時間が大幅に異なってきます。次のプログラムで計算時間を見てみましょう：

- ファイル名：**combinat2.py**

```
1   from time import time    #timeモジュールからtimeという関数をインポート
2
3   def kaijou(n):
4       if n == 0 or n==1:
5           return 1
6       else:
7           return n*kaijou(n-1)
8
9   def C(m,n):     # (1)で組み合わせ数を定義
10      return kaijou(m)//(kaijou(m-n)*kaijou(n))
11
12  def D(m,n):     # (2)で組み合わせ数を再帰的に定義
13      if n<0 or m<n:
14          return 0
15      elif m == 0 or m==n:
16          return 1
17      else:
18          return D(m-1,n) + D(m-1,n-1)
19
20  time1 = time()  # この時間をtime1とする
21  print(C(25,12))    # C(25,12)を計算
22  time2 = time()    # このときの時間をtime2とする
23  print('計算に要した時間は', time2-time1, '秒')
24  print(D(25,12))    # D(25,12)を計算
25  time3 = time()    # このときの時間をtime3とする
26  print('計算に要した時間は', time3-time2, '秒')
```

実行結果の例

```
5200300
計算に要した時間は 2.6941299438476562e-05 秒
5200300
計算に要した時間は 1.5428187847137451 秒
```

式 (3) を使って計算した場合の計算時間は約 0.000027 秒なのに対して，(4) を使った計算では 1.54 秒かかっています。

なぜ，後者で 10000 倍も計算時間が必要だったのでしょうか？それは漸化式 (4) を使った計算では非常に遠回りをして答えを出しているからです。例えば，D(5,2) の計算ですら，次のような膨大な計算を行っている

からです：

$$\begin{aligned}
D(5,2) &= D(4,2) + D(4,1) \\
&= D(3,2) + D(3,1) + D(3,1) + D(3,0) \\
&= D(2,2) + D(2,1) + D(2,1) + D(2,0) + D(2,1) + D(2,0) + D(2,0) + D(2,-1) \\
&= 1 + D(1,1) + D(1,0) + D(1,1) + D(1,0) + D(1,0) + D(1,-1) + D(1,1) + D(1,0) \\
&\quad + D(1,0) + D(1,-1) + D(1,0) + D(1,-1) + 0 \\
&= 1 + 1 + D(0,0) + D(0,-1) + 1 + D(0,0) + D(0,-1) + D(0,0) + D(0,-1) + 0 + 1 \\
&\quad + D(0,0) + D(0,-1) + D(0,0) + D(0,-1) + 0 + D(0,0) + D(0,-1) + 0 \\
&= 1 + 1 + 1 + 0 + 1 + 1 + 0 + 1 + 0 + 0 + 1 + 1 + 0 + 1 + 0 + 0 + 1 + 0 + 0 \\
&= 10
\end{aligned}$$

こんなやり方では，もっと大きな組み合わせの数 ${}_{54}C_{24} = 1402659561581460$ を計算するのに約 100 年ぐらいかかってしまうでしょう。ですから，コンビネーションを与える関数を定義するには公式 ${}_mC_n = m!/(m-n)!n!$ を使うべきです。このように簡単に再帰的に定義できる関数でも，闇雲に使うと事実上計算不可能なことになるかもしれないので注意が必要です。

次にフィボナッチ数列

$$a_1 = 1,\ a_2 = 1, \qquad a_{n+2} = a_{n+1} + a_n, \qquad n = 1, 2, \cdots$$

を例に取ってみましょう。次は与えられた自然数 `n` に対して a_n を返す関数 `fib(n)` を定義したものです。

- ファイル名：**fib1.py**

```
1 def fib(n):
2     if n==1 or n==2:
3         return 1
4     else:
5         return fib(n-1) + fib(n-2)
6
7 for j in range(1,40):
8     print(fib(j))
```

実行結果の例

```
1
1
2
...
39088169
63245986
```

上のプログラムを実行すると，j が増えるにしたが値が出るのが遅くなるのがわかると思います。これは `fib(n)` の値が 39088169 であれば，39088168 回もの和を計算しているからです。したがって，フィボナッチ数列を再帰的に定義するのはうまいやり方ではないことがわかると思います。

しかし，少し工夫をして一度計算した関数の値を憶えておくことにより，計算の高速化を行うことができます。このような方法を**メモ化** (memoization) といいます。`fib(n)` に対するメモ化の手順には辞書[*8] を使います。辞書 `memo` には `n` と対応する `fib(n)` の値を

$$\texttt{memo = \{1:1, 2:1, 3:2, 4:3, 5:5, 6:8, 7:13, 8:21, ...\}}$$

のように記録しておくことにしましょう。

(1) 初期状態の辞書 `memo = {1:1, 2:1}` を作る。
(2) 関数 `fib(n)` の定義を開始：もし鍵 `n` が `memo` の中にないなら，辞書 `memo` に `n:fib(n-1)+fib(n-2)` を記録する。そして `memo[n]` に対応する値を返す。

[*8] 辞書は `s = {3:'a', 5:'b', 6:'c'}` のようなデータで `3:a` の 3 をキー，`'a'` をその値と呼ぶことを思い出しましょう。また，`s[7]='d'` とすることで，項目 `7:'d'` を辞書に追加することができます：`s = {3:'a', 5:'b', 6:'c', 7:'d'}`

上の手順で fib1.py をメモ化してみましょう。

- ファイル名：**fib2.py**

```
memo = {1:1, 2:1}

def fib(n):
    if n not in memo: # キー nが辞書にないなら
        memo[n] = fib(n-1)+fib(n-2)  # 辞書に項目を追加
    return memo[n]

for j in range(1,40):
    print( fib(j) )
```

実行結果の例

```
1
1
2
...
39088169
63245986
```

上のプログラムでは，fib(3)，fib(4) と順次計算を進めるに従い，その値が memo に記録されていきます。したがって，例えば fib(30) を計算するときには，fib(29)，fib(28) の値がすでに memo の中にあるので，初期条件 fib(1),fib(2) まで戻ることなく，関数の値が求められるというわけです。

注意 関数の最大再帰回数は Python3 ではデフォルトで 1000 に設定されているので，上のプログラムでは fib(1000) は計算できますが，fib(1001) ではエラーとなります。関数の定義より前に次を書いておくことで，再帰回数の上限を 2000 回に変更することができます：

```
import sys
sys.setrecursionlimit(2000)
```

16.8 練習問題

気の利いた問題ではないのですが，とりあえず練習問題です。

16.8.1 再帰的な定義の練習

和

$$f(n) = \sum_{k=1}^{n} 4\frac{(-1)^{k+1}}{2k-1}$$

は，$f(1) = 4$, $f(n) = f(n-1) + \frac{4(-1)^{n+1}}{2n-1}$ を満たすことを利用してこれを再帰的に定義なさい。

- ファイル名：**sum1.py**

```
def f(n):
   *******:
      ****** *
   *****
      ****** ****************************

for j in range(1,1000, 100):
    print(f(j))
```

実行結果の例

```
4
3.1514934010709914
3.1465677471829556
3.1449149035588526
3.144086415298761
3.143588659585789
3.143256545948974
3.1430191863875865
3.142841092554028
3.1427025311614294
```

16.8.2 メモ化の練習

上の問題の関数をメモ化して定義して，同様の計算をしなさい。ファイル名は sum2.py とすること。

17　その他の便利な機能

17.1　リストの操作とリスト内包表記

リストを定義する際に，それが短いものであれば，次のように直接的に書くことで定義することができます。

```
>>> a = [1,3,2,4]
>>> a
[1,3,2,4]
```

リストに要素を付け加えるには，append メソッドを使いました：

```
>>> a.append(7)
>>> a
[1,3,2,4,7]
```

リストをある種のパターンで生成するためには，for 文などの繰り返しの処理を使って要素を付け加えます。次の例では，空のリストに i^2 $(i = 0, 1, 2, \cdots, 9)$ を付け加える事によって，平方数からなるリストを生成しています。

```
aa = []                          # aaを空のリストとする
for i in range(10):              # iを0から9まで動かして
    aa.append(i*i)               # i*iをリストaに付け加える。

print(aa)                         # aaをプリントする
```

実行結果の例

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

次に $1, 9, 25, 49, 81, \cdots$ と奇数の平方数からなるリストを for 文を使って作ってみましょう。

```
bb = []                           # bbを空のリストとする
for i in range(20):               # iを0から19まで動かして
    if i%2 == 1:                  # もしiが奇数なら
        bb.append(i*i)            # i*iをリストaに付け加える

print(bb)                         # bbをプリントする
```

実行結果の例

```
[1, 9, 25, 49, 81, 121, 169, 225, 289, 361]
```

Python にはリスト内包表記という非常に便利なリストの生成法があります。リスト内包表記を使えば，上のようなリストをたった一行で作ることができます。リスト内包表記を使って上と同じリストを作って表示させるプログラムがこちらです：

```
cc = [i*i for i in range(10)]      # リスト内包表記
print(cc)
```

実行結果の例

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

結果は上と同じですが，リストを簡潔に定義することができました。さらにリスト内包表記で要素を生成するときに**条件を付ける**こともできます。上で作ったリスト bb は次のように簡潔に作ることができます。

```
dd = [i*i for i in range(20) if i%2==1]
print(dd)
```

実行結果の例

```
[1, 9, 25, 49, 81, 121, 169, 225, 289, 361]
```

さらに，次のように二つの変数を動かして要素を生成することもできます：

```
lis = ['a', 'b', 'c']
ee = [ (i,j) for i in range(1,5) for j in lis]
print(ee)
```

実行結果の例

```
[(1, 'a'), (1, 'b'), (1, 'c'), (2, 'a'), (2, 'b'), (2, 'c'),
 (3, 'a'), (3, 'b'), (3, 'c'), (4, 'a'), (4, 'b'), (4, 'c')]
```

17.2　名前のない関数 — lambda 式

lambda 式と呼ばれる特別な文法があり，名前がなくて引数と返り値の対応だけを指定して関数を定めることができます。これは対応だけを指定したいが，わざわざ名前を付けたくないときに用います。そして Python では `lambda` は特別な意味を持つ予約語なので変数名などで使ってはいけません。

lambda 式の文法

```
lambda x, y : (xとyを使った式)
```

引数と戻り値の間はコロン：で区切ります。

例えば n に n^2 を対応させる関数 f を lambda 式を使って次のように書くことができます：

```
>>> f = lambda a : a*a     # fは引数aに対してa*aを返す関数
>>> f(3)
9
```

また，lambda 式はリストの要素になることが出来ます。

```
a = [lambda x, y : x+y,    lambda x,y: x-y,    lambda x,y: x*y]

for f in a:
    print(f(3,5))
```

実行結果の例

```
8
-2
15
```

17.3　ライブラリ・モジュール

Python には標準ライブラリと呼ばれる部品の集まりがあります。標準ライブラリには，特定の用途ごとに使う関数がモジュールという塊で管理されています。

17.3.1　math モジュール

数学でよく使う定数や関数が定義されたものが math モジュールです。

- ファイル名：**math1.py**

```
import math    # mathモジュールをインポート
```

```
print(math.pi)  # mathにあるpiを表示
print(math.e)   # mathにあるeを表示
print(math.sin(1))   # mathにある関数sinの1における値を表示
```

実行結果の例

```
3.141592653589793
2.718281828459045
0.8414709848078965
```

もちろん，これらは数学の $\pi, e, \sin(1)$ の数値 (近似値, float) です。また，次のように使用する定数・関数だけをインポートすることで math. を書く手間を省くことができます。

- ファイル名：**math2.py**

```
from math import pi, sin  # mathモジュールのpiとsinをインポートする
print(pi)           # 単にpi
print(sin(1))       # やsin( )と書くだけで円周率の値や
print(sin(pi/2))    # sin関数を利用できる。
```

実行結果の例

```
3.141592653589793
0.8414709848078965
1.0
```

数学関数はよく使うものだし，個別に関数をインポートするのも面倒なので，次のように math モジュールにあるすべての関数をインポートするがよいでしょう。

- ファイル名：**math3.py**

```
from math import *  # mathモジュールのすべての関数をインポート
print(pi)           # 円周率の近似値
print(sin(1))       # sin( )の近似値
print(log10(2))     # 対数関数も使える
```

実行結果の例

```
3.141592653589793
0.8414709848078965
0.3010299956639812
```

math モジュールで定義されている関数の種類は，次の Python の公式ページから確認できます：

https://docs.python.org/ja/3/library/math.html

17.3.2　random モジュール

1 から 1000 までの数字 x をランダムに生成して，x を 10 回以内で当てるゲームを作りたい。

乱数とは特定の分布を持つがパターンのない数列のことです。コンピュータのように決定論的に動作する機械では真の乱数を作ることはできませんが，実際上乱数とみなせるような数列を作ることができます。そのような乱数を擬似乱数といいます。Python ではライブラリをインポートすることで手軽に（疑似）乱数を使うことができます。次が乱数を使った簡単なプログラムの例です。

```
>>> import random   # 乱数のrandomモジュールを使う
>>> x = random.randint(1,100)   # xを1から100までのランダムな数にする
>>> x
>>> 73     # 人によって出力は異なります。
```

一回だけではわからないので，何回も繰り返し乱数を発生させてみましょう。

- ファイル名：**random1.py**

```
from random import randint   #randomモジュールの関数randintを使う
```

```
2 for j in range(20):
3     print(randint(1,100))
```

実行結果の例

```
37 84 100 21 32 26 73 69 21 66 10 79 40 70 33 23 76 26 44 69
```

上の結果では random モジュールを用いて，1 から 100 までの数が無作為に 20 回取り出されて表示されました。

さて，次のような手順を行う数字当てゲームを作りたい。

(i) 1 から 1000 までの数を一つランダムに生成し，それを x とする。
(ii)『x を当ててみよう』と表示する。
(iii) キーボードから入力された数字を変数 a に入れる。
(iv) $a = x$ なら『正解』と表示しゲームは終了する。
(v) $a < x$ なら『もっと大きい』と表示し，$a > x$ なら『もっと小さい』と表示する。
(vi) 上の (iii)–(v) を 10 回繰り返し，当てることができなければ，『Game Over』と表示する。

より詳細な手順を示したアルゴリズムはつぎの通りです。

(1) ライブラリ `random` をインポートする。
(2) 1 から 1000 までの数を一つランダムに生成し，それを x とする。
(3)『1 から 1000 までの数字 x がランダムに生成されました。x を 10 回以内で当ててみよう！』と表示する。
(4) for 文で j を 0 から 9 まで変えながら，次の (5), (6) の処理を繰り返す。
(5) `input` で『残り $10 - j$ 回です。x は何でしょう？：”』と表示して，キーボードからのデータを取得し，それを (整数に変換してから) 変数 a に入れる。
(6) もし $a = x$ なら『正解』と表示し，`break` で for 文から抜け，
もし $a < x$ なら『もっと大きいよ』と表示し，
そうでないなら『もっと小さいよ』と表示する。
(7) (for 文が終わった後に) もし，$x \neq a$ なら次の (8), (9) を行う。
(8)『正解は x でした』と表示する。
(9)『残念 Game Over』と表示する。

上のアルゴリズムを Python プログラムとして書いてみましょう。ファイル名は `find_number.py` としてください。

【補足】 PCR 検査の方法に，複数の検体を混合し同時に検査する，プール検査というものがあります。例えば 5 人分の検体をまとめて PCR 検査した結果が陽性なら 5 人の中に 1 名以上はウイルスに感染している人がいるということになります。上のプログラムは 1000 人の中に 1 人だけ感染者がいると仮定した場合，10 回以内のプール検査で陽性者を発見しようというものと考えることもできます。(ただし，偽陰性が無いと仮定した場合です。混合する検体数が増えれば偽陰性の割合も増えてしまいます。厚生労働省のページによればプール検査は 5 検体を推奨しているそうです (2021 年)。)

18　Set 型の操作と応用

18.1　基本的な集合の操作

以前少し紹介しましたが Python には集合 (set) というデータの型があります。これはデータの集まりであることはリストと同じですが，順番がない事と重複が除かれる事がリストとは異なる所です。数学で扱う集合と同じですが，もちろん扱えるのは有限集合だけです。集合は {1,3,4} のように表します。集合同士の演算『union ∪，intersection ∩』をそれぞれ『|, &』で行うことができます。例えば，Python のインタラクティブシェルでの集合の操作は次のようにします：

```
>>> a = {1,3,4}      # 集合 a を定義
>>> b = {3,7}        # 集合 b を定義
>>> c = a | b        # c を集合 a と b の和集合とする
>>> c
{1, 3, 4, 7}
>>> d = a & b        # d を集合 a と b の共通部分とする
>>> d
{3}
```

集合の演算を行うには，union() や intersection() というメソッドを使うこともできます。また|=を使うことで，集合に他の集合の要素を加えることができます：

```
>>> a.union(b)          # a.union(b) は
{1, 3, 4, 7}            # a と b の和集合を返す
>>> a.intersection(b)   # a.intersection(b) は
{3}                     # a と b の共通部分を返す
>>> a
{1, 3, 4}               # 上の操作で a の値が変わるわけではない
>>> b
{3, 7}                  # b もかわらない
>>> a |= b              # a に b の要素をすべて追加する
>>> a                   # a には
{1, 3, 4, 7}            # b の要素が追加される
```

集合に要素を追加するには add，要素を削除するには remove を使います：

```
>>> a.add(8)            # 集合 a に要素 8 を加える
>>> a                   # a の値を聞く
{1, 3, 4, 7, 8}         # a に要素 8 が追加された
>>> a.remove(1)         # a から要素 1 を除く
>>> a
{3, 4, 7, 8}            # a から 1 が削除された
```

Python3 では辞書も set と同じように {1:4, 2:9, 3:11} のように中括弧で書く事を思い出しましょう。{} は空の辞書 (dict) なので気をつけなければなりません。空集合は set() で表わします。

```
>>> a = {}
>>> type(a)
<class 'dict'>      # {} は辞書です。
>>> a = {1}
>>> a.remove(1)
```

```
>>> a
set()                    # 空集合はset()と記述する
```

18.2　Set 型の応用——四則演算で 10 を作る遊び

鉄道の切符に書かれている 4 桁の数字（または自動車のナンバーの 4 つの数字）から四則演算で 10 を作るゲームがあります。例えば，切符の数字が 2339 なら，

$$(2+9)-(3/3)=10$$

のようにして数字 10 を作ることができます。4 つの数字はどのような順番で計算してもかまいません。しかし 1111 のような数字からはどうやっても 10 を作ることは出来ません。どのような数字の組なら四則演算で 10 を作ることが出来るでしょうか？ 10 を作ることができるような数字の組を列挙するプログラムを Python で書いてみたいと思います。これを行うプログラムはいろいろな方法で書けると思いますが，ここでは set を使ってみます。（ちなみに切符の端に書かれている 4 桁の数字は 0 をとらないらしいですが，ここでは 0 を含む場合も考えます。）

18.2.1　2 項演算

まずは 2 つの数字の組 a,b の四則演算で作られる数 $a+b, a-b, b-a, a*b, a/b, b/a$ を要素に持つ集合を返す関数を作ります。これを nikou(a,b) としましょう。この場合，得られる数の順番や重複は意味がないので，set を使うのが便利なのです。割り算のときに 0 で割る可能性を排除するために場合分けが必要です：

- ファイル名：**nikou.py**

```
def nikou(a,b):
    if a !=0 and b!=0:      # aもbも0でないときは
        return {a+b, a-b, b-a, a*b, a/b, b/a} # 四則演算からできる集合
    elif b == 0:            # b=0のときは
        return {a, -a, 0}
    else:                   # a=0のときは
        return {b, -b, 0}

print(nikou(3,4))     #3,4から四則演算で作ることができる数の集合を表示
```

実行結果の例

```
{0.75, 1, 7, 12, 1.3333333333333333, -1}
```

18.2.2　3 つの数の四則演算

つぎに，3 つの数 a,b,c の四則演算で作られる数の集合を返す関数 sankou(a,b,c) を作りたいと思います。まず 3 つの数はどの順番で計算するかによって 3 通りの順番があることに注意します。ある二項演算を $\circledast, \ominus$（つまり $\circledast, \ominus$ は＋－×÷のどれか）として

- $(a \circledast b) \ominus c$　：　a,b を先に計算してから次に c との演算を行う。
- $(a \circledast c) \ominus b$　：　a,c を先に計算してから次に b との演算を行う。
- $(b \circledast c) \ominus a$　：　b,c を先に計算してから次に a との演算を行う。

のように 3 つの計算の順番があります。

上のことに注意して 3 つの数 a,b,c に対する 2 項演算から作られる数の集合を返す関数 sankou(a,b,c) を定義してみましょう。

- ファイル名：**sankou1.py**

```
def nikou(a,b):
    if a !=0 and b!=0:
        return {a+b, a-b, b-a, a*b, a/b, b/a}
    elif b == 0:
        return {a,-a,0}
    else:
        return {b, -b, 0}

def sankou(a,b,c):        # a,b,cの四則演算で作られる集合を返す
    ResultSet = set()     # ResultSetを空の集合とする
    for i in nikou(a,b):        # a,bから作られる数iに対して
        ResultSet |= nikou(i,c) # i,cの二項演算から作られる集合をResultSetに追加
    for i in nikou(b,c):
        ResultSet |= nikou(i,a)
    for i in nikou(a,c):
        ResultSet |= nikou(i,b)
    return ResultSet       # ResultSetを返す

print(sankou(4,4,9))          # 4,4,9を1回ずつ使った四則演算で作られる集合
print('')                     # 改行
print(10 in sankou(4,4,9))    # sankou(4,4,9)には10は入っているか？
```

実行結果の例

```
{0.8888888888888888, 1, 1.125, 0, 0.5625, 1.7777777777777777, 0.1111111111111111, 7, 8.0,
 9, 10.0, 3.5555555555555554, 4.444444444444445, 1.75, 6.25, 1.25, 144, 17, 3.25, 20, 25,
 32, 40, -0.8, 52, 72, 0.8, 0.3076923076923077, -32, -20, -9, -8.0, -7, -1.75,
 -3.5555555555555554, -1, -1.25}

True
```

さて，これで 3 つの数字の組から 10 を作れるかどうかを判定する事が可能になりました。そこで三つの数字 $0 \leq a \leq b \leq c \leq 9$ で四則演算によって 10 を作ることができるものをすべて列挙してみましょう：

- ファイル名：**sankou2.py**

```
def nikou(a,b):
    if a !=0 and b!=0:
        return {a+b,a-b,b-a,a*b, a/b, b/a}
    elif b == 0:
        return {a,-a,0}
    else:
        return {b, -b, 0}

def sankou(a,b,c):       # a,b,cの四則演算で作られる集合を返す
    ResultSet = set()    # ResultSetを空の集合とする
    for i in nikou(a,b):        # a,bから作られる数iに対して
        ResultSet |= nikou(i,c) # i,cの二項演算でできる集合をResultSetに追加
    for i in nikou(b,c):
        ResultSet |= nikou(i,a)
    for i in nikou(a,c):
        ResultSet |= nikou(i,b)
    return ResultSet     # ResultSetを返す

counter = 0      # counterという変数を作り0にセットする

```

```
21 for i in range(10):
22     for j in range(i,10):
23         for k in range(j,10):
24             if 10 in sankou(i,j,k):    # もし10がsankou(i,j,k)の要素なら
25                 print(i,j,k)           # i,j,kを表示して
26                 counter = counter + 1 # numberを1増やす
27
28 print('10を作れる数字の三つ組みの数は', counter, '個')
```

実行結果の例

```
0 1 9
0 2 5
...
...
9 9 9
10 を作れる数字の三つ組みの数は 75 個
```

18.2.3 4 つの数字の四則演算

つぎに 4 つの数 a,b,c,d から作ることのできる数の集合を返す関数を作ります。どれか 2 つの数を先に計算することで，3 つ組の場合に帰着します。例えば，a,b を先に計算してその計算結果と c,d との四則演算で作ることのできる数の集合を作るには，a,b のすべての計算結果 i に対して sankou(i,c,d) を作ればよいです。a,b,c,d のうちどの 2 つを先に計算するかについては，${}_4C_2 = 6$ 通りの場合があります。つまり，最初に計算する数は全部で (a,b),(a,c),(a,d),(b,c),(b,d),(c,d) の 6 通りです。このような手順で作られる数字の集合を返す関数を yonkou(a,b,c,d) とします。この関数は次のように定義します。

```
1  def yonkou(a,b,c,d):
2      ResultSet = set()
3      for i in nikou(a,b):              # a,bの計算結果iに対して
4          ResultSet |= sankou(i,c,d)    # i,b,cから作られる数をResultSetに追加
5      for i in nikou(a,c):
6          ResultSet |= sankou(i,b,d)
7      for i in nikou(a,d):
8          ResultSet |= sankou(i,b,c)
9      for i in nikou(b,c):
10         ResultSet |= sankou(i,a,d)
11     for i in nikou(b,d):
12         ResultSet |= sankou(i,a,c)
13     for i in nikou(c,d):
14         ResultSet |= sankou(i,a,b)
15     return ResultSet
```

結局，0 から 9 までをとる 4 つの数字 a, b, c, d で $a \leq b \leq c \leq d$ かつこれらの四則演算で 10 を作ることができる組をすべて列挙するプログラムを作成するには次のようにします：

- ファイル名：**make10.py**

```
1  def nikou(a,b):
2      if a !=0 and b!=0:
3          return {a+b, a-b, b-a, a*b, a/b, b/a}
4      elif b == 0:
5          return {a,-a,0}
6      else:
7          return {b, -b, 0}
8
```

```
9  def sankou(a,b,c):          # a,b,cの四則演算で作られる集合を返す
10     ResultSet = set()       # ResultSetを空のリストとする
11     for i in nikou(a,b):          # a,bから作られる数iに対して
12        ResultSet |= nikou(i,c)  # i,cの演算から作られる集合をResultSetに追加
13     for i in nikou(b,c):
14        ResultSet |= nikou(i,a)
15     for i in nikou(a,c):
16        ResultSet |= nikou(i,b)
17     return ResultSet          # ResultSetを返す
18
19 def yonkou(a,b,c,d):
20     ResultSet = set()
21     for i in nikou(a,b):              # a,bの計算結果iに対して
22         ResultSet |= sankou(i,c,d)  # i,b,cから作られる数をResultSetに追加
23     for i in nikou(a,c):
24         ResultSet |= sankou(i,b,d)
25     for i in nikou(a,d):
26         ResultSet |= sankou(i,b,c)
27     for i in nikou(b,c):
28         ResultSet |= sankou(i,a,d)
29     for i in nikou(b,d):
30         ResultSet |= sankou(i,a,c)
31     for i in nikou(c,d):
32         ResultSet |= sankou(i,a,b)
33     return ResultSet
34
35 counter = 0
36
37 for i in range(10):
38     for j in range(i,10):
39        for k in range(j,10):
40           for l in range(k,10):
41              if 10 in yonkou(i,j,k,l):
42                  print(i,j,k,l)
43                  counter = counter+1
44
45 print('10を作れる数字の4つ組みの数は', counter, '個')
```

実行結果の例

```
.......
.......
.......
8 8 9 9
9 9 9 9
10 を作ることができる 4 つの数字の組の数は 552 個
```

上のプログラムは結果的にはうまく動いていますが，本来なら桁落ちに注意が必要です。二項演算を浮動小数点で行っているために，演算の結果が正確には 10 のはずなのに，誤差により 10 ではなくなっている可能性があるからです。ただ，10 に非常に近い数は 10 であると認識されます：

```
1 >>> 9.9999999999999999 == 10
2 True
3 >>> 9.999999999999999 == 10
4 False
```

18.3 練習問題

5 つの数字 $a, b, c, d, e (0 \leq a \leq b \leq c \leq d \leq e \leq 9)$ を 1 回ずつ使い四則演算（加算，減算，乗算，除算）で 100 を作ることを考える。100 を作れるような全ての数の組及びその個数を表示するプログラムを作成せよ。ファイル名は `make100.py` とすること。

19　GUI アプリの作り方

Python には標準で Tkinter というグラフィカルなソフトを作るためのツールが用意されています。ここでは，Tkinter を使って簡単な GUI プログラムを作る方法を紹介します。

19.1　Tkinter の基本

Tkinter は Python のモジュールです。次のようにインポートして使います。

- ファイル名：**gui1.py**

```
from tkinter import *

win = Tk() # ウィンドウを一つ定義
win.geometry("400x200")  # 窓のサイズ（横x縦）

moji = Label(win, text='こんにちは')
moji.pack()  # 窓に配置する

## ウィンドウを開始
win.mainloop()
```

Tkinter にはウィジェット（Widget，部品）が色々用意されていて，上の `Label` はウィジェットの一つです。

次のようにして，ウィンドウのタイトルを付けたり，文字の枠サイズを指定したりできます。

- ファイル名：**gui2.py**

```
from tkinter import *

win = Tk() # ウィンドウを一つ定義
win.geometry("400x200")  # 窓のサイズ（横x縦）
win.title("あいさつ")

label = Label(win, text='こんにちは')
label.pack(padx=20, pady=20)  # 枠サイズを指定

## ウィンドウを開始
win.mainloop()
```

19.2　ボタンの設置

ボタンはウィジェットとして用意されていますが，まずボタンを押したときの動作を関数で書いておく必要があります。そして Button ウィジェットを定義するときに，その関数を指定します。

- ファイル名：**gui3.py**

```
from tkinter import *

win = Tk()
win.geometry("400x200")  # 横 x縦

# ボタンの動作を表す関数を定義
def hello():
    print('なますて〜')

btn1 = Button(win, text="click me", command=hello) # ボタンを定義
btn1.pack() # ボタンを詰める

win.mainloop() # ウィンドウを開始
```

上を実行すると，「click me」と書かれたボタンのあるウィンドウが表示され，ボタンを押すとターミナル (CUI) の方に「なますて〜」とプリントされます。

次に，ターミナルではなくウィンドウに挨拶文が表示されるようするには次のようにします。

- ファイル名：**gui4.py**

```
from tkinter import *

win = Tk() ### メインのウィンドウの定義
win.geometry("400x200")  # ウィンドウサイズ
win.title('あいさつ')

aaa = Label(win, text = "No Data") ### 表示する文字の定義


def india(): # ボタン1の動作
    aaa["text"] = 'なますて〜'
def japan(): # ボタン2の動作
    aaa["text"] = 'こんにちは〜'

btn1 = Button(win, text="click me(india)", command=india) # ボタン1を作る
```

```
16 btn1.pack()
17
18 btn2 = Button(win, text="click me(japan)", command=japan) # ボタン2を作る
19 btn2.pack()
20
21 aaa.pack() ### aaaを配置
22
23 win.mainloop() ### ウィンドウを開始
```

ボタンを押すと，ウィンドウ内に「なますて〜」や「こんにちは〜」と表示されます。

次のようにしてテキストを文字列を削除するボタンを作ったりすることができます。

- ファイル名：**gui5.py**

```
1  from tkinter import *
2
3  win = Tk() # メインのウィンドウの定義
4  win.geometry("400x200")  # ウィンドウサイズ
5  win.title('あいさつ')
6
7  aaa = Label(win, text = "No Data") # 表示する文字の定義(まだpackしない)
8
9  def india(): # インドのあいさつ
10     aaa["text"] = 'なますて〜'
11 def japan(): # 日本のあいさつ
12     aaa["text"] = 'こんにちは〜'
13 def clear_aaa(): # aaaの文字を削除する関数
14     aaa["text"] = ""
15
16 btn1 = Button(win, text="click me(india)", command=india)
17 btn1.pack()
18 btn2 = Button(win, text="click me(japan)", command=japan)
19 btn2.pack()
20 btn3 = Button(win, text="クリア", command=clear_aaa)
21 btn3.pack()
22
23 aaa.pack() # aaaを配置
24
25 win.mainloop() # ウィンドウを開始
```

19.3　文字列の取得

画面に文字を入力する箇所を作ったり，そこから文字列を取得したりするには Entry，StringVar ウィジェットを使います。

- ファイル名：**gui6.py**

```
from tkinter import *

win = Tk() ### メインのウィンドウの定義
win.geometry("400x200")  #ウィンドウサイズ
win.title('入力文字の取得')

aaa = Label(win, text='なまえ') ### テキストウィジェットを定義
aaa.pack() # 配置する

bbb = Label(win, text='') ### ラベルウィジェットを定義

### エントリー（入力する場所）
data0 = StringVar()
ccc = Entry(win, textvariable= data0)
ccc.pack()

# 関数を定義
def getname():
    hh = ccc.get()    # 入力された文字列を取得し変数hhに入れる
    bbb["text"] = "こんにちは"+hh+"さん！" # bbbの文字を変更

btn1 = Button(win, text="btn1", comman=getname) # ボタンを定義
btn1.pack()

bbb.pack()  # bbbを配置

win.mainloop() ### ウィンドウを開始
```

19.4　簡単な計算機の作成

入力されたものを計算するだけの簡単な GUI 計算機を作ります。文字列を式にして実行するには eval 関数を用います。これは

```
>>> moji = '2+3'
>>> eval(moji)
5
```

のように動作します。次は入力されたものを計算して表示するプログラムです。

- ファイル名：**gui7.py**

```
from tkinter import *

win = Tk() ### メインのウィンドウの定義
win.geometry("400x200")  # ウィンドウサイズ
win.title('簡単な計算機')

aaa = Label(win, text='計算式') # ラベルを定義
aaa.pack()

### エントリー（入力する場所）
date0 = StringVar()      # 文字列を入れる変数
bbb = Entry(win, textvariable= date0) # エントリーの定義
bbb.pack()  # つめる

ccc = Label(win, text='') # 計算結果を表示する部分

# 関数を定義
def jikko():
    hh = bbb.get() # 入力情報を取得
    hh = str(eval(hh)) # 計算
    ccc["text"] = "計算結果:"+hh # 結果の出力

def clearentry():
    bbb.delete(0, "end") # エントリーのクリア

btn1 = Button(win, text="RUN", comman=jikko) # RUNボタン
btn1.pack()
btn2 = Button(win, text="Clear", command=clearentry) #Clearボタン
btn2.pack()

ccc.pack() # 計算結果を表示するウィジェットをつめる

win.mainloop() ### ウィンドウを開始
```

math ライブラリをインポートしておけば，sin, log などの計算もできるようになります。

- ファイル名：**gui8.py**

```
from tkinter import *
from math import *

win = Tk() # メインのウィンドウの定義
win.geometry("400x200")  # ウィンドウサイズ
win.title('簡単な計算機')

aaa = Label(win, text='計算式') # ラベルを定義
aaa.pack()

# エントリー（入力する場所）
data0 = StringVar()
bbb = Entry(win, textvariable= data0)
bbb.pack()

ccc = Label(win, text='') # 計算結果を表示する部分

# 関数を定義
def jikko():
    hh = bbb.get() # 入力情報を取得
    hh = str(eval(hh)) # 計算
    ccc["text"] = "計算結果:"+hh # 結果の出力

def clearentry():
    bbb.delete(0, "end") # エントリーのクリア

btn1 = Button(win, text="RUN", comman=jikko) # RUNボタン
btn1.pack()
btn2 = Button(win, text="Clear", command=clearentry) #Clearボタン
btn2.pack()

ccc.pack() # 計算結果を表示するウィジェットをつめる

win.mainloop() ### ウィンドウを開始
```

第 II 部
数式処理システム SageMath

20　数式処理システム SageMath とは

SageMath(以下 Sage と略) は代数・幾何・数論・数値計算等の広範囲の数学をサポートする数式処理プログラムです。実際の数学の研究でも利用されています。Sage は Python で書かれていて，Sage のプログラムも Python の文法で書きます。Sage を使うことによって，行列の計算，数値計算，組み合わせ論，特殊関数，微分方程式の解法といった様々な数学的な計算を手軽に行うことができます。また，Sage はフリーでオープンソースなソフトウェアですので，だれでも自由に入手して使うことが可能です。Linux，Mac，Windows，オンラインで利用可能です[*9]。

20.1　Sage の公式ページ

Sage の公式ウェブサイトはこちらです。

`https://www.sagemath.org/`

日本語の情報はほとんどありませんが，上のページから次の情報にアクセスできます：

- CoCalc(CoCalc Instant SageWorksheet) or SageMathCell：インターネットから Sage 利用することができます。
- Download：Sage プログラムのダウンロード (インストール方法は OS によって異なります)。
- Help/Documentation： Sage や Python に関する説明書やチュートリアル。解説ビデオなどもあります。
- Sage Feature Tour：いろいろな活用方法が紹介されています。

Sage の日本語版のチュートリアルはこちらにあります。

`http://doc.sagemath.org/pdf/ja/tutorial/tutorial-jp.pdf`

初心者向けに丁寧に解説されており，手短に Sage の機能を体験することができます。

20.2　Sage の実行方法

Sage を実行する方法はいくつかありますが，Sage がコンピューターにインストールされているのであれば，以下の方法で Sage を実行することができます：

1. インタラクティブ・シェル (端末から `sage` で起動)
2. ファイルから実行 (端末から `sage ファイル名.sage`)
3. Sage ノートブックを使う (端末から `sage -n` で起動)

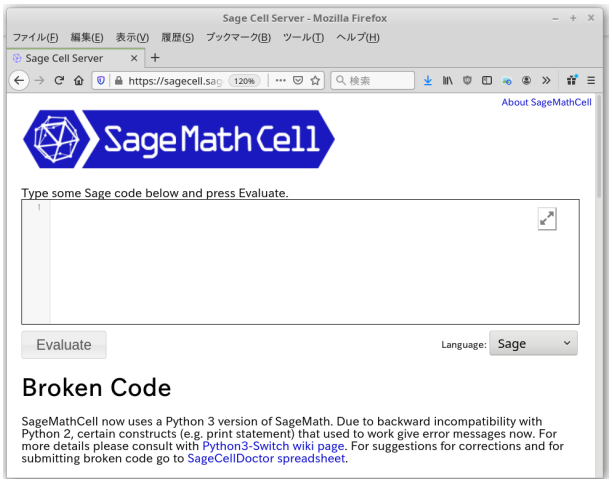
[*9] しかし，Sage のプログラムサイズは年々増加しており，SageMath9.3(Linux) のインストールファイルは圧縮状態で 2.8GB にもなりました。今後はオンラインでの利用が一般的になっていくかもしれません。

20.3　オンラインでの Sage の利用 (SageMathCell)

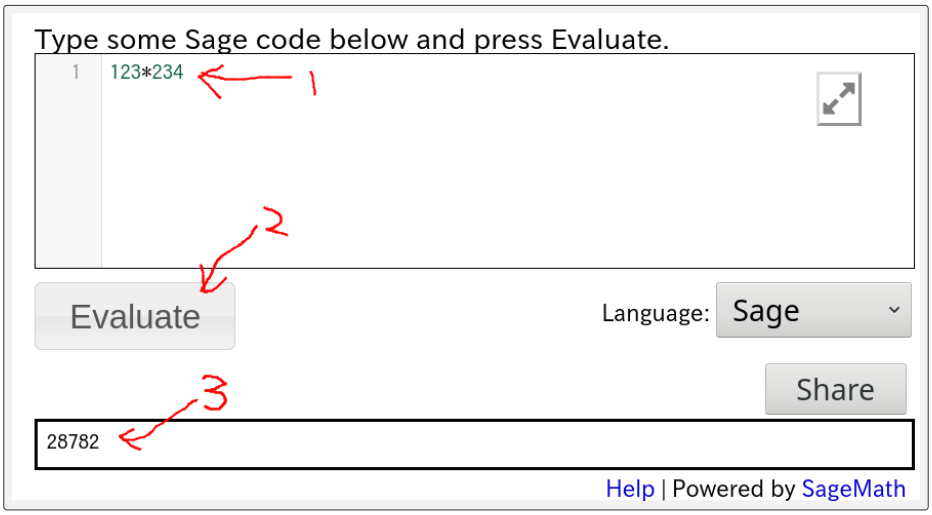
Sage がコンピューターにインストールされていなくても，インターネットに接続されていればオンラインの Sage サービスを利用して計算することができます。最も簡単で簡易的なものは SageMathCell を利用することです。これには Sage のトップページの (CoCalc の下にある) リンクをクリックするか直接

`https://sagecell.sagemath.org/`

にアクセスすることで次のようなページが開き，直ちに Sage の簡易的な計算ができるようになります。



ここで計算するにはこの中央の枠にプログラムを書いて左下にある [**Evaluate**] ボタンをクリックします。計算結果はその下に表示されます。



SageMathCell には保存の機能はないので，プログラムと実行結果はメモ帳などに貼り付けて保存します。

20.4　Linux での Sage の利用（インタラクティブシェル）

Linux Mint に Sage がすでにインストールされているものとします。端末を開き sage と入力すると Sage のインタラクティブシェルが起動します。

```
mint@mint-bv:~$ sage
 ┌────────────────────────────────────────────────────────────────────┐
 │ SageMath version 9.x, Release Date: 202x-xx-xx                     │
 │ Using Python 3.9.5. Type "help()" for help.                        │
 └────────────────────────────────────────────────────────────────────┘
sage: factor(1111)
11 * 101
sage: exit
Exiting Sage (CPU time 0m0.21s, Wall time 1m16.12s).
mint@mint-bv:~$
```

上では `factor(1111)` で 1111 の素因数分解を行っています。Sage インタラクティブシェルを終了するには `exit` とタイプするか `CTRL+C` を押します。

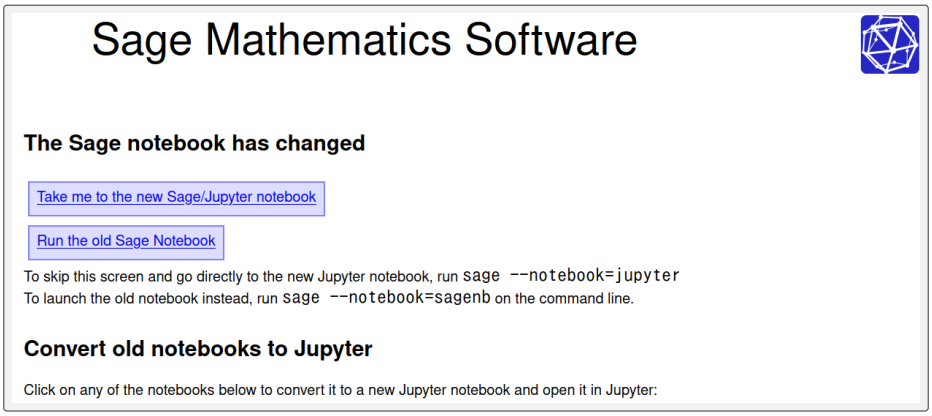
20.5 Linux での Sage の利用（ノートブック）

Linux Mint に Sage がすでにインストールされているものとします。Jupyter ノートブックを使うことで，手軽に計算結果を表示したりグラフを描画したりすることができるようになります。

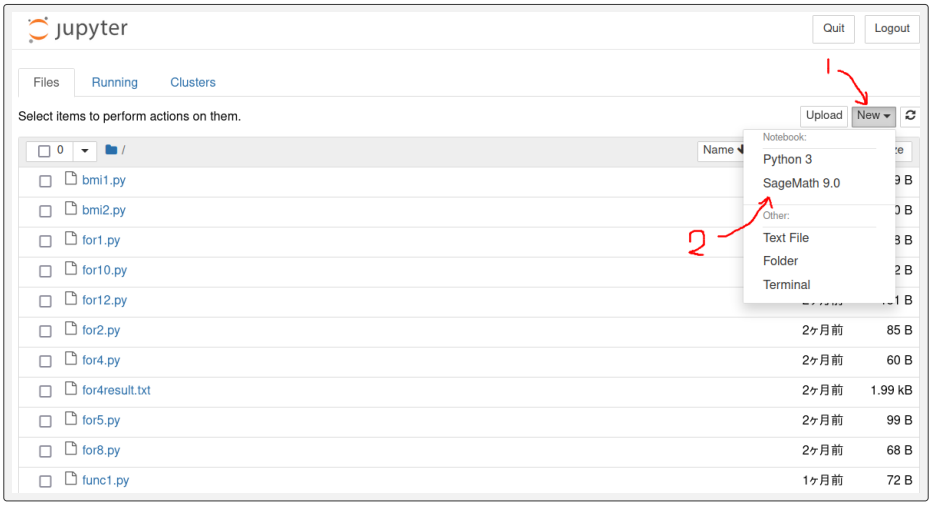
端末を開き，ディレクトリを変更してから `sage -n` とオプション付きで Sage を起動します：

```
mint@mint-vb:~$ cd Desktop/dataproc1
mint@mint-vb:~/Desktop/dataproc1$ sage -n
```

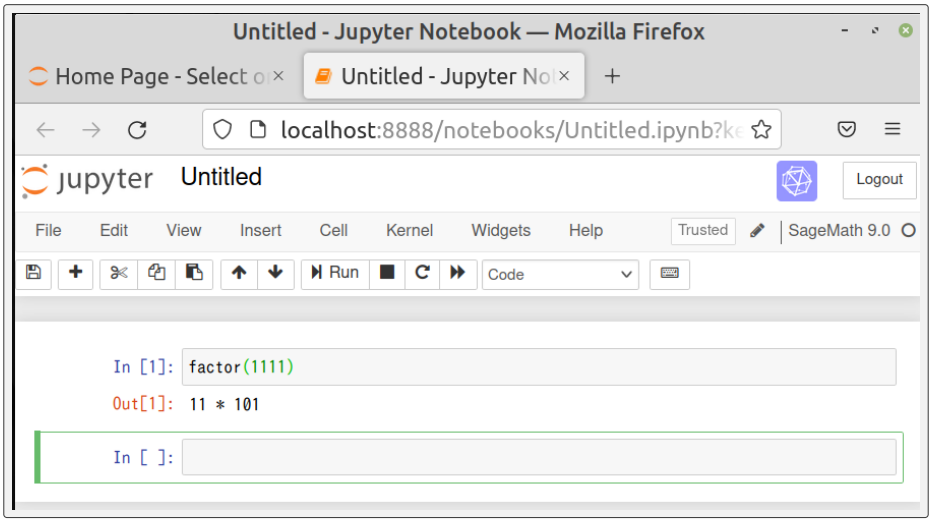
するとブラウザ (Firefox) が起動し，次のような画面になります：



「Take me to the new Sage/Jupyter notebook」をクリックすると，次のような dataproc1 のディレクトリの中身の一覧が表示されます。



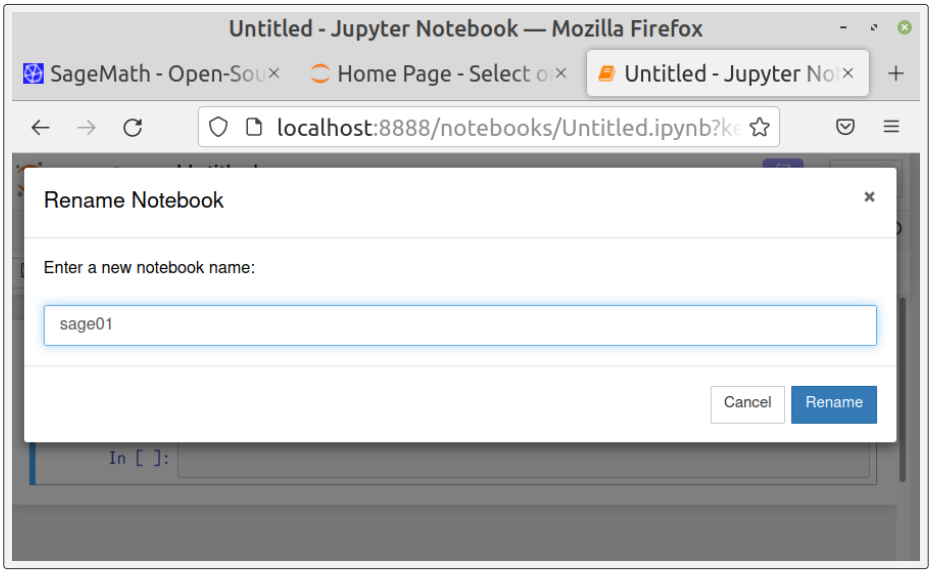
まだ Jupyter ノートブックが無いので作ります。右上の「New」をクリックし，「SageMath 9.x」をクリックします。すると次のような Jupyter ノートブックの画面になります。画面にあるセルに `factor(1111)` と入力して `Shift+Enter` を押せば，`1111` が因数分解された結果が，セルの下に表示されます。



まだ，この Jupyter ノートブックには名前は付けていないのですが，起動したディレクトリ（いまは Desktop/dataproc1）に Untitled.ipynb というファイルが作られています。

20.5.1 ファイル名変更と保存

Untitled.ipynb がデフォルトの Jupyter ノートブックのファイルですが，これを Jupyter ノートブックとして開いている状態で，左上の「File」→「Rename...」とクリックして名前の変更をします。



ここでは新しいファイル名を Sage01 とします。画面上の Unititled と書かれた部分が sage01 に変わったことを確認してください。画面左上のフロッピーディスクのマーク[*10]をクリックすることで，ノートブックの保存ができます。Sage ノートブックを終了するにはブラウザを閉じて，端末で CTRL+C を押します。すると Shutdown the notebook server y/[n]?などと聞かれるので y Enter を押します。

20.5.2　ノートブックを開く

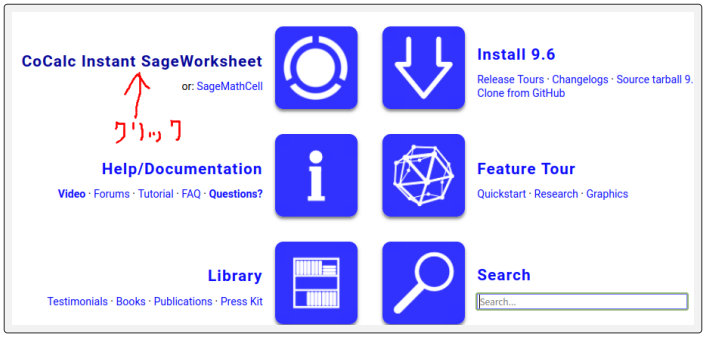
上に解説した手順でノートブックを作った場合，dataproc1 フォルダの中に sage01.ipynb が作られていると思います。このファイルをダブルクリックしても Jupyter ノートブックは開かないので注意してください。

ノートブックを開くには，まず 20.5 節で説明したのと同じように，端末からディレクトリを移動した後に Sage を起動します。そしてディレクトリにあるファイルの一覧が表示されたら sage01.ipynb をクリックすることで，Jupyter ノートブックを読み込むことができます。

20.6　オンラインでの Sage の利用 (CoCalc & jupyter)

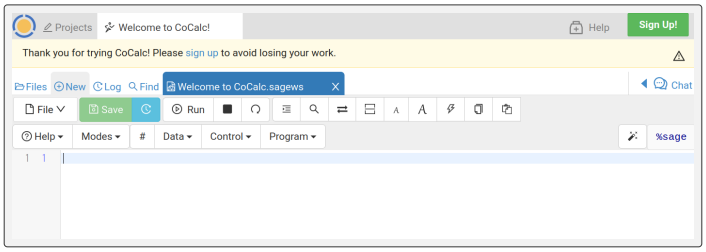
20.6.1　アクセスから実行まで

CoCalc はクラウドで計算を行うプラットフォームです。ここでは CoCalc で動作する Jupyter ノートブックというものから Sage を利用する方法を説明します。SageMath の公式ページにある CoCalc Instant SageWorksheet と書かれたリンクをクリックして CoCalc のページに移動しましょう。



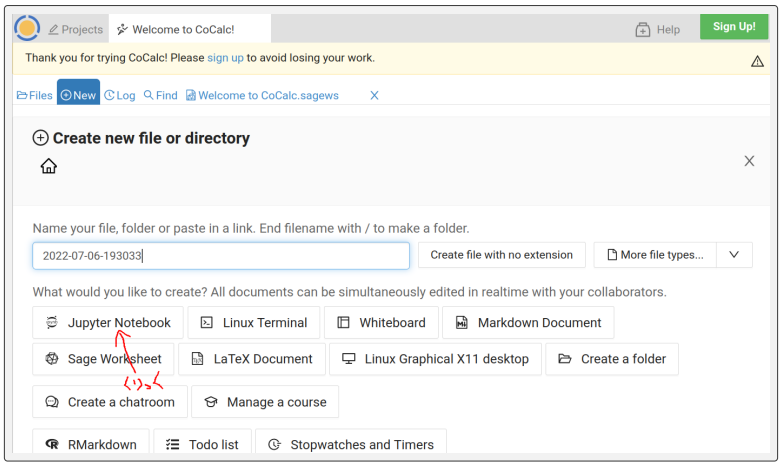
[*10] といっても今の人は知らないか・・

CoCalc の初回の画面は次のようになります[*11]：



上の画面でも Sage を動かすことはできます。しかし，この計算シートは Sage ワークシート形式という今後廃止される予定のものですので，この解説では，別の形式である Jupyter ノートブックを紹介します。

上の画面の左上の方にある「⊕ New」をクリックすると次のような画面になります。



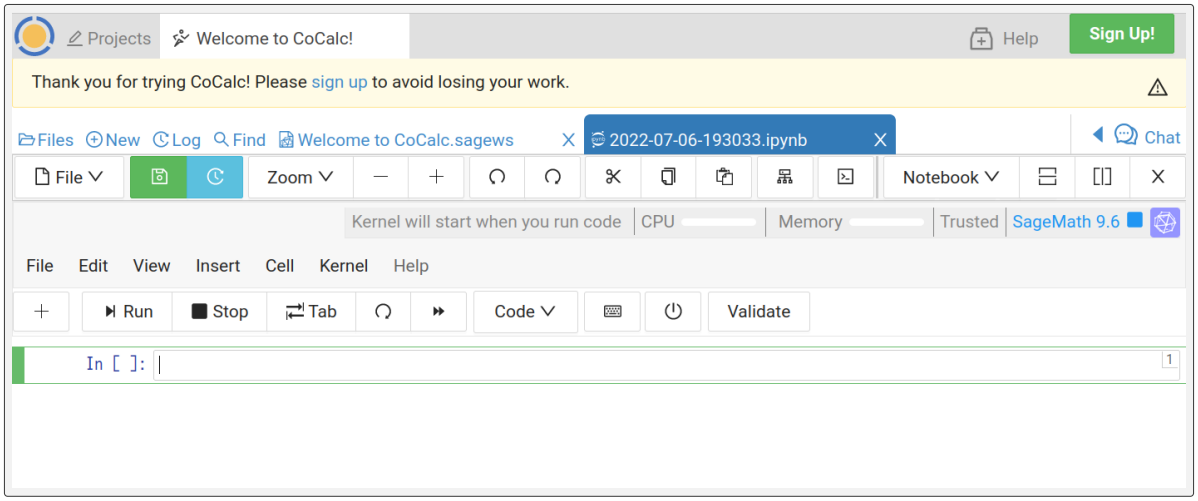
次に「Jupyter Notebook」をクリックしてください。次のようなカーネルの選択画面になります。

この環境では Julia や Python 3，R，SageMath[*12]などを計算カーネルに設定することができます。いまは，「SageMath 9.6」[*13]を選択します。すると次のような画面になります。

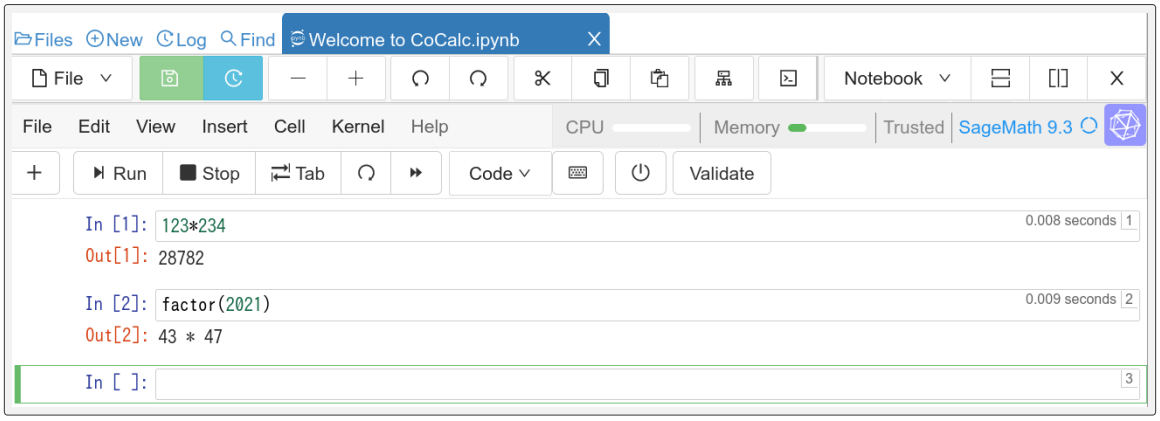
[*11] 2 回目以降は上の画面にならないかもしれませんが，その場合は (i)Cookie を削除する (ii) プライベートブラウジングモードにする (iii) ブラウザを再起動する（iv）履歴を削除するなどを試してしてみてください。

[*12] Julia は数値計算用のプログラミング言語，R は統計処理ソフトウェア。

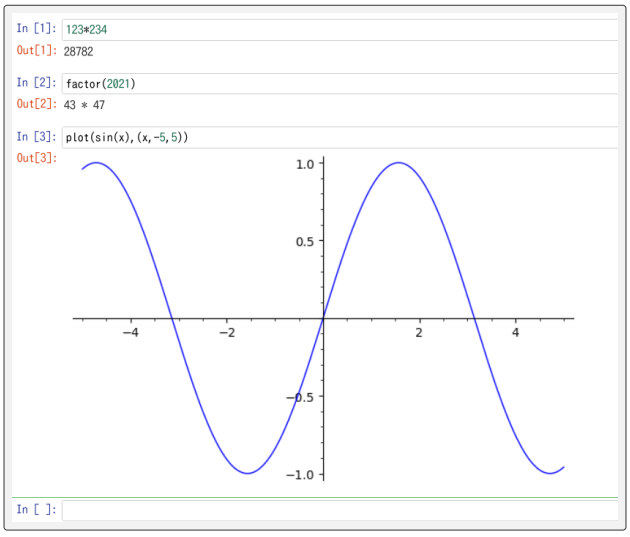
[*13] バージョンは変わる可能性があります。



これで CoCalc で SageMath の計算を行う準備が整いました。`In [ ]:`の左にある枠をセルといい，ここにプログラムを書きます。セルに書いたプログラムを実行するには `Shift+[Enter]` とタイプするか，`Run` と書かれたボタンをクリックします。するとセルの下に実行結果が表示され，新しいセルが作られます。`123*234` と `factor(2021)` を実行した結果が次の画像です：



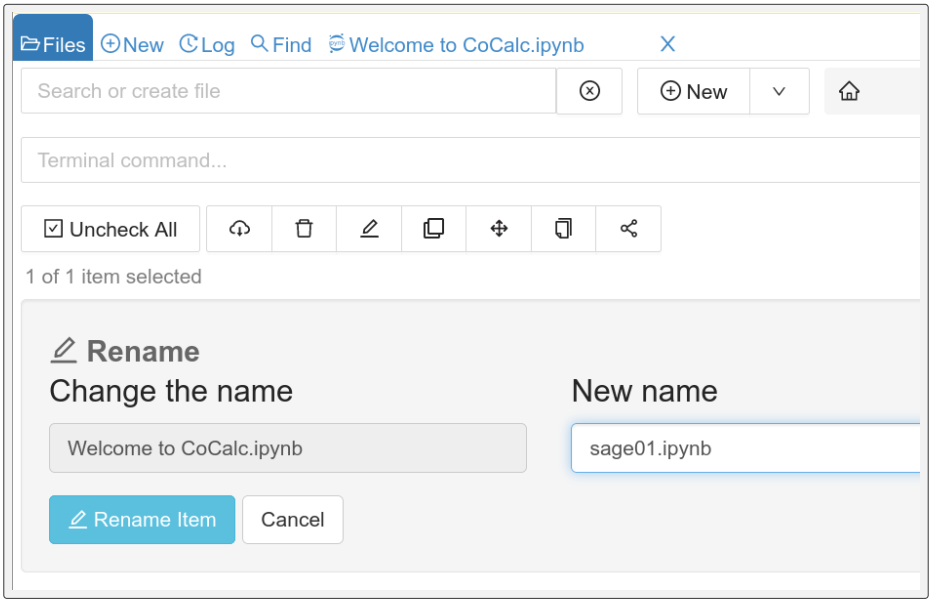
`factor(2021)` は 2021 を素因数分解する命令です。また，`plot(sin(x),-5,5)` と入力することで $\sin x$ のグラフを表示させることもできます。



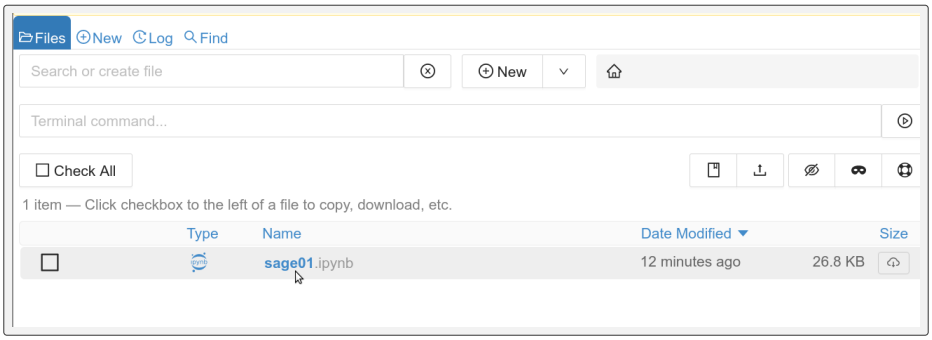
20.6.2　ノートブックの保存

この Jupyter ノートブックをファイルとして保存する方法を紹介します。上部の青いタブに`***.ipynb` と表示されています (ただし，`***`の部分は人によって異なります) が，これが現在の Jupyter ノートブックのファイル名になります。

まずはファイル名の変更をしましょう。左上の「`File`」から `Rename` をクリックします。New name の下の枠に新しいファイル名を入力します。ここではファイル名は `sage01.ipynb` とします[*14]。

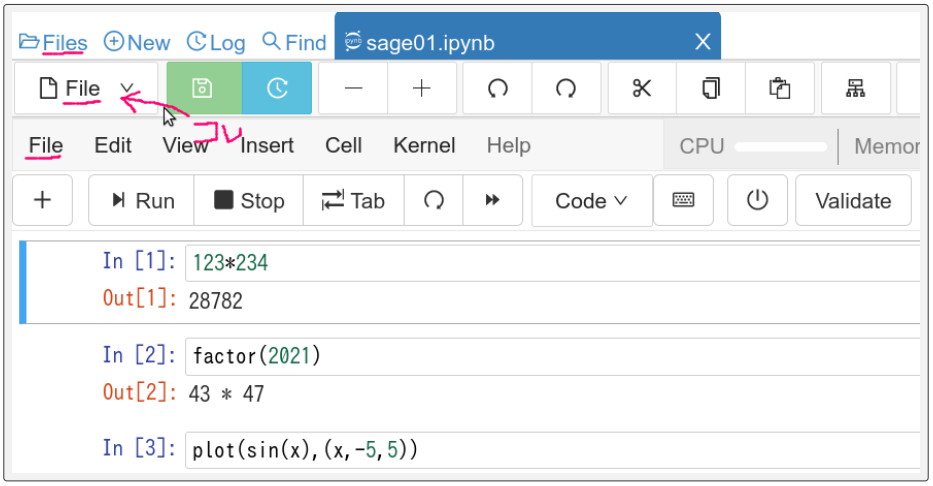


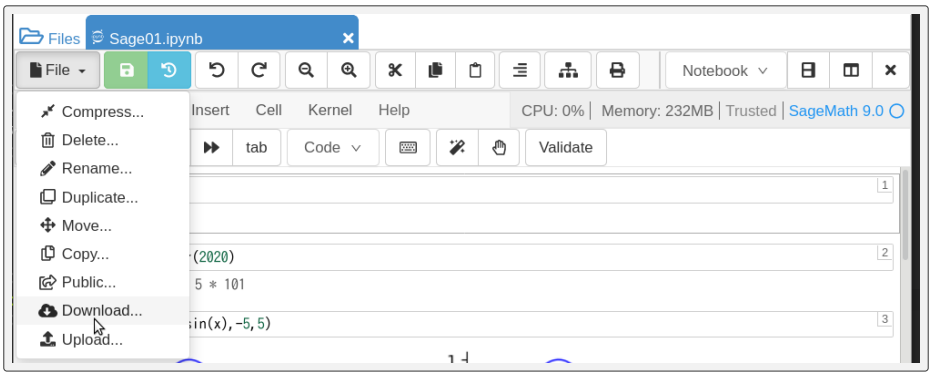
[Rename Item] をクリックするとノートブックのファイル名が変更されます。**sage01**.ipynb をクリックすると元のワークスペースに戻ります。更新中は 20 秒ほど待ちましょう。



上部の青色のタブのファイル名が `sage01.ipynb` に変わったことを確認しましょう。次にこのノートブックを保存します。画面の左側には 3 つの「Files」,「File」,「File」という項目があり，分かりづらいのですが，2 番めの File をクリックし，**Download...** を選択します。

[*14] 拡張子 (`.ipynb`) を変更してはいけません



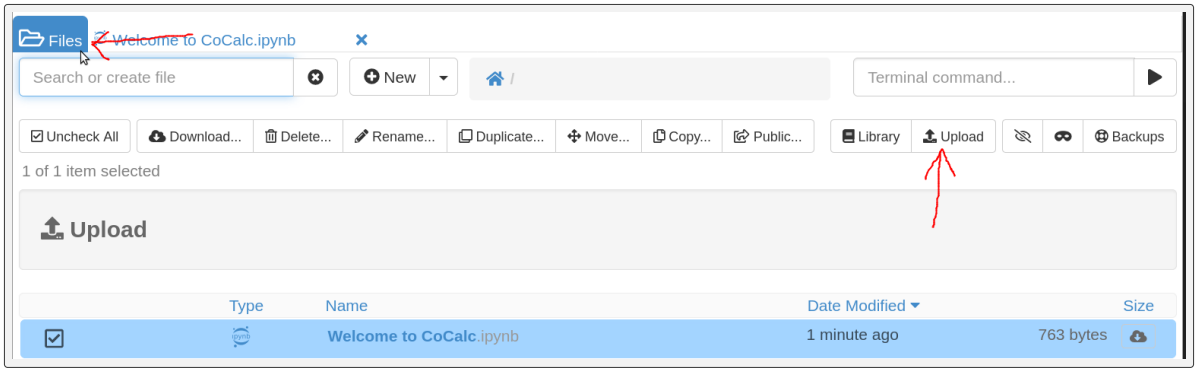


左下の **Download** ボタンをクリックするとノートブック (sage01.ipynb) がダウンロードされます。

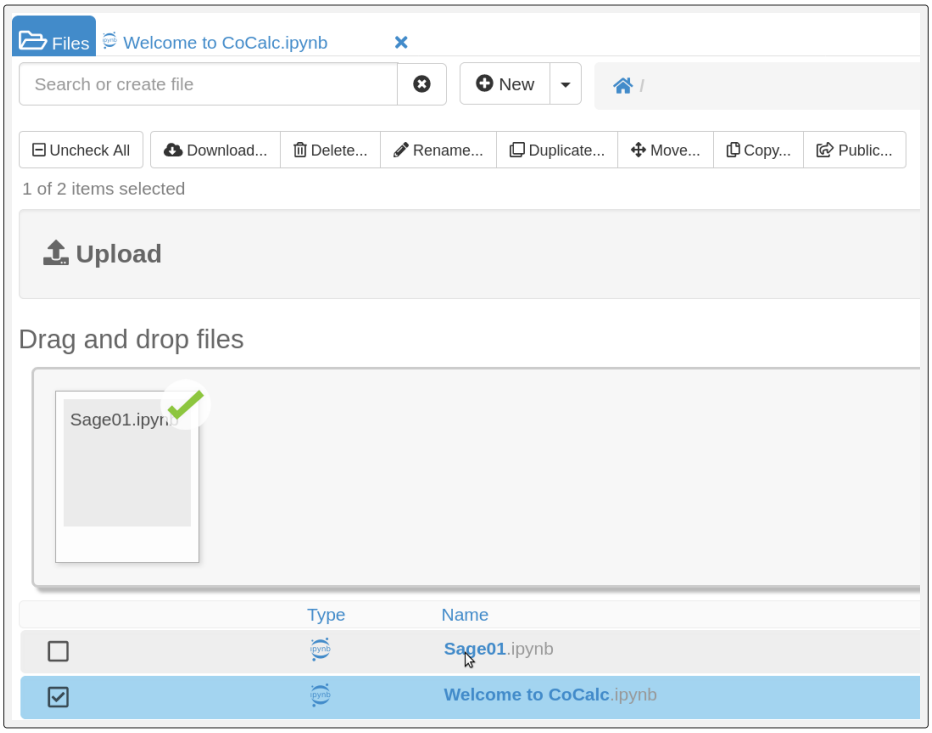
ダウンロードされるファイルの場所はブラウザの設定により異なりますが，Windows で無設定の場合は「ダウンロード」フォルダに保存されます。ダウンロードしたファイル `sage01.ipynb` をデスクトップにある `dataproc1` フォルダに移動させておきましょう。

20.6.3　保存したファイルの読み込み

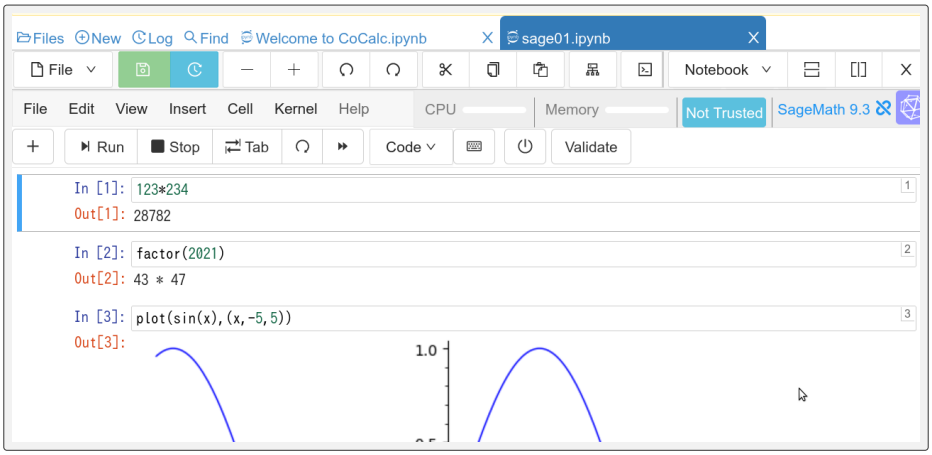
次に保存したノートブックファイル (sage01.ipynb) を読み込む方法を解説します。一旦ブラウザを終了し，再度 CoCalc を起動します。先ほどと同様に CoCalc が起動したら，左上の Files タブをクリックします。



上の画面のように **Upload** をクリックしてから，保存してある `sage01.ipynb` を選びます。



上の図のようにアップロードができたら，**sage01**.ipynb をクリックすることで，保存した時と同じノートブックが表示されます。



21　Sage の計算の基本

前節で説明したように Sage プログラムを実行する方法はいくつかありますが，以下では特定の実行方法に依存しないような解説にするために，一行ごとに Sage を実行 (評価，evaluate という) する事を

```
sage: 1+2
3
sage: factor(2010)
2 * 3 * 5 * 67
```

のように説明します。ここでプログラムを試すときには sage:以下を書きます。まとまった Sage プログラムを実行する場合には

```
a = sqrt(2)
b = sqrt(8)
print(a*b)
```

実行結果の例

```
4
```

と説明します。この場合，出力は最後に評価された結果，もしくは print されたものになります。

前節から続けて読んでいる人は，これから説明するプログラムを sage01.ipynb のセルに書いて評価してみてください。

21.1　四則演算

Sage の四則演算は，基本的には Python と同じで

$$a+b \to \texttt{a+b} \qquad a-b \to \texttt{a-b} \qquad a\times b \to \texttt{a*b} \qquad \frac{a}{b} \to \texttt{a/b} \qquad a^b \to \texttt{a\^{}b}$$

ですが，割り算の記号『/』の扱いが異なります。整数 a, b に対して Python3 では a/b は商を返しましたが，Sage では形式的な有理数 a/b となります。

```
sage: 5/4              # 分数は
5/4                    # 分数のまま扱うことが可能
sage: 5.0/4            # 少数点数があれば
1.25000000000000       # 商が計算される
sage: 10/5
2
sage: 4/5 + 5/6    # 有理数の足し算
49/30              # 通分されて計算される
```

これからわかるように，有理数同士の四則演算は数学的に厳密な結果を返します。

冪の記号は Python では a^b は a**b でしたが，Sage では『a^b』と表わすこともできます。

```
sage: 2^10
1024
sage: 2**10
1024
```

21.2 数の表示

Sage では整数，有理数，有限体等を取り扱うことができ，無理数も形式的に取り扱うことができます。円周率 π は pi，自然対数の底 e は e，虚数単位 i は i で表されます。次の命令を順次実行して確認してみましょう：

```
sage: pi            # 円周率
pi                  # そのまま
```

円周率の数値を表示するには n() という命令を使います：

```
sage: n(pi)
3.14159265358979    # 円周率の近似値
sage: pi.n()        # このように書いても同じ
3.14159265358979
sage: n(pi, digits=30)   # 円周率を30桁表示
3.14159265358979323846264338327950288 4197
```

自然対数の底についても同様です。

```
sage: n(e)
2.71828182845905
```

平方根 (square root) は sqrt(3) のように表わします：

```
sage: sqrt(3)   # √3
sqrt(3)
sage: sqrt(3)*sqrt(3)
3
sage: n(sqrt(3))
1.73205080756888
```

Sage では文字 n に，n(a) で『a の数値を返す』という関数があらかじめ割り当てられているので，変数名や関数名に文字 n を使わないようにしましょう。

21.2.1 複素数

Sage では虚数単位 i は I で表され，複素数 $5+3i$ は 5+3*I のように表します。複素数の計算は実数の場合と同じように行うことができます。

```
sage: (5+3*I)^2
30*I + 16
sage: a = 5+3*I;  a.conjugate()  # 複素共役
-3*I + 5
sage: e^(pi*I)
-1                  # Eulerの公式が使える
```

21.2.2 数学定数

Sage ではじめから定義されている数学定数は e や π の他には次のようなものがあります (ほとんど使うことはありませんが…)。

名前	数学記号	Sage の記法	定義
黄金比 (golden ratio)	ϕ	`golden_ratio`	$\frac{1+\sqrt{5}}{2}$
オイラーの定数 (Euler's constant)	γ	`euler_gamma`	$\lim_{n\to\infty}\left(\sum_{k=1}^{n}\frac{1}{k}-\ln(n)\right)$
カタランの定数 (Catalan's constant)	K	`catalan`	$\sum_{k=1}^{\infty}\frac{(-1)^n}{(2n+1)^2}.$
双子素数の定数 (twin prime)	C_2	`twinprime`	$\prod_{p\geq 3;\ 素数}\frac{p(p-2)}{(p-1)^2}$

```
sage: n(golden_ratio)
1.61803398874989
sage: n(twinprime)
0.660161815846870
```

21.2.3 数の精度

数値計算をするときにはどの精度で計算するのかを気にする必要があります。Sage で取り扱うことができる代表的な数の型に次のようなものがあります。

名前	Sage の記号	Sage の英語名	意味	精度
整数	`ZZ`	Integer Ring	整数のつくる環	厳密
有理数	`QQ`	Rational Field	有理数体	厳密
代数的数	`QQbar`	Algebraic Field	$\mathbb{Q}$ の代数閉包	厳密
形式的な環	`SR`	Symbolic Ring	形式的な環（ほぼ体）	厳密
実数 (53bit)	`RR`	Real Field with 53 bits of precision	53 ビットの実数	近似
実数 (53bit)	`RDF`	Real Double Field	倍精度浮動小数点実数	近似
複素数 (53bit)	`CC`	Complex Field with 53 bits of precision	53 ビットの複素数	近似
実数 (400bit)	`RealField(400)`		400 ビットの実数	近似

これらは必要に応じて使い分けます。高精度で計算すると少ない誤差で計算できますが，多くの計算時間がかかります。上記以外にも $\mathbb{Z}/p\mathbb{Z}$ や $\mathbb{Q}[\sqrt{5}]$ などさまざまな環や体を取り扱うことができます。以下を実行して確かめてみましょう：

```
sage: ZZ
Integer Ring        # 名前が出てきます。QQ,RR,SR等でも同様
sage: 5 in ZZ       # 5は整数か？
True
sage: 1.5 in ZZ
False
sage: 1.5 in QQ     # 1.5は有理数か？
True
```

```
sage: sqrt(2) in QQ     # ルート2は有理数ではない
False
sage: sqrt(2) in QQbar # ルート2は代数的数である
True
sage: x in SR         # xは形式的に取り扱える文字か？
True

sage: y in SR         # yは形式的に取り扱える文字か？
......                                  # エラーメッセージ
NameError: name 'y' is not defined  # yは定義されていない
```

上の実行結果の最後のように，文字 x は形式的に扱える数 SR としてデフォルトで定義されていますが，y は定義されていません。

円周率の精度は次のようになります：

```
sage: RDF(pi)
3.14159265359
sage: RR(pi)
3.14159265358979
sage: RealField(200)(pi)
3.1415926535897932384626433832795028841971693993751058209749
```

次にやっかいな誤差が生じる場合を紹介します。

```
sage: a = RR(pi)     # aは円周率の近似値
sage: exp(a*I)       # e^(iπ) = -1
 -1.00000000000000 + 1.22464679914735e-16*I
```

$e^{i\pi} = -1$ なのですが，上の実行結果では，非常に小さい虚数部分 1.22464679914735e-16*I が残っています。これは数値計算から生じる誤差です。答えが実数でないために，それ以降のプログラムにエラーが生じることがあります。例えば実数値関数のグラフ描画 (plot) を行うときに関数の値に虚部ががあると，それがどんなに小さくても実軸上にプロットできずエラーが生じます。複素数値を含む関数の数値計算を取り扱う場合は，このようなことを常に注意しておかなければなりません。

21.3 Sage のヘルプ

命令の後に『?』を書いて実行すると，その命令の使い方が表示されます。次を実行してみましょう：

```
factor?
```

因数分解を返す関数 factor の使い方とその例が表示されたはずです。さらに

```
factor??
```

とすることで関数 factor を定義しているソースコードが表示されます（が，普通は見る必要はありません）。

22　Jupyter ノートブックの機能の説明

Jupyter ノートブックで Sage の計算を行う方法やノートブックの保存，読み込みの方法は前に説明しました。ここでは Jupyter の機能をいくつか紹介します。ここでは Jupyter notebook のことを単に notebook といいます。

22.1　カーネルの初期化

notebook の裏では Sage のカーネル (計算を行うプログラム) が動いており，一つのセルで `a=3` を実行すると，Sage カーネルは変数 `a` が 3 であるという情報を持ち続けます。次に `a=5` と入力すれば `a` の値は 5 に変わります。Sage カーネルが記憶している変数の情報をリセットするには notebook 上部にある🔄 を押すか，『Kernel』→『Restart Kenel』を実行します。

Do you want to restart the kernel? All variables will be lost.
カーネルを再起動したいですか？ すべての変数は失われます。

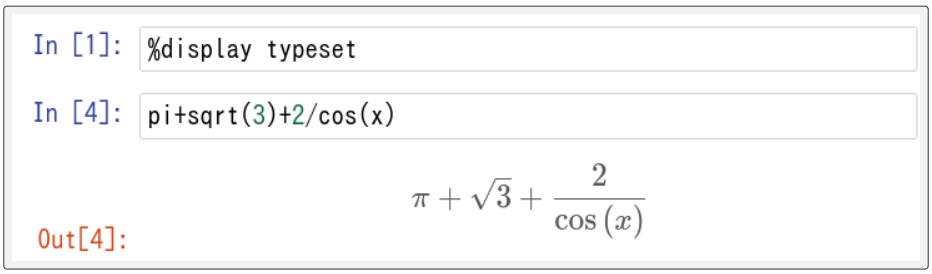
と警告が出ますが，**Restart** を押すことでカーネルが初期化されます。

22.2　補完機能

端末や Python インタラクティブシェルでは，プログラム名や関数名などを途中まで入力して `Tab` キーを押すとキーワードが自動的に補完されますが，Jupyter notebook も同様の機能を持っています。セルで `fac` と入力してから `Tab` キーを押してみましょう。`factor`(因数分解) と `factorial`(階乗) の 2 つの候補が出てきます。

22.3　出力の LaTeX による整形

Jupyter notebook のセルに `%display typeset` と書いて実行してすると，それ以降の実行結果が**数学の書式で整形**された形で出力されます。



上の図にしたがって，一つのセルで `%display typeset` を実行してから，`pi+sqrt(3)+2/cos(x)` を実行してみてください。上の図のように LaTeX でタイプセットされた数式が表示されます。タイプセットを解除するには `%display plain` とします。

23　代数的計算

Sage や Mathematica, Maple といった計算機代数システム (CAS) の大きな特徴の一つが，文字式を扱うことができ，式の変形や微分や積分などの代数的計算を行うことができる点です。CAS を使いこなすために，

文字式や関数と呼んでいたものと，以前に解説した Python における関数を明確に区別する事から始めます。

23.1 Python の関数と Sage の関数

Sage では Python の関数とは異なる関数（=文字式）があるわけですが，まずは Python における関数の復習をします。Python における関数は，いくつかの処理をひとかたまりにして名前を付けたものでした。たとえば，次のように関数を定義しました：

```
1  def fib(n):                  # 関数fibの定義，引数n
2      a, b = 0, 1              # ここから
3      for i in range(n):       #
4          a, b = b, a+b        # ここまでが一連の処理
5      return a                 # 上の処理が終わったらaの値を返す
6
7  for i in range(1,10):            #i=1,…,9 に対して
8      print(fib(i), end=' ')       # fib(i)をプリント
9
10 print(type(fib))             # fibのデータの型を表示
```

実行結果の例

```
1 1 2 3 5 8 13 21 34
<type 'function'>
```

上のプログラムでは，関数 `fib()` が呼び出されると，そのときの引数 `n` に対して 2 行 ~4 行の処理を行い，それが終わると `a` の値を返す (`return`) という事をしています。

はじめて Python の関数を習ったときに違和感を憶えた人も多いと思います。それは，高校以前の数学で『関数』と呼ばれているものは**文字式**として定義されていて，方程式を解いたり微分したりといった操作ができるものだったからだとおもいます。たとえば

$$4x+3, \quad x^2, \quad \sin(x), \quad \frac{1}{x^2+1} \tag{5}$$

はどれも文字式でもあり関数でもあります。ここでは，x は関数の変数 (variable) や不定元 (indeterminate) と呼ばれます。これらの関数は，x に具体的な数値を代入すれば，計算によりその関数の値が定まりますが，Python の関数のように計算手順を記述したものではありません。x は多くの場合実数ですが，状況によっては複素数や行列かもしれません。

Sage では『文字式としての関数』に相当するものを取り扱うことができます。`x` は最初から形式的な文字として取り扱うことができます。

```
1 sage: x in SR  # xは形式的な文字式か
2 True
3 sage: x+sin(x) in SR  # x+sin(x)は文字式か
4 True
```

これを文字 `y` でやってみるとエラーとなります。

```
1 sage: y in SR
2 ...
3 ... name 'y' is not defined   # エラーメッセージ
```

`x` 以外の文字を文字式として使うためには，次のように宣言をする必要があります。

形式的な文字の定義

```
sage: var('y')    # y を文字式に使う
y
sage: y in SR
True              # y は文字式に使える
```

文字式による関数は次のように定義します。

文字式としての関数の定義 1

$f = x^2$ の定義は

```
f = x^2           # x^2 を変数 f に入れる
```

$g = \sin y$ の定義は

```
var('y')          # y を文字式に使う
g = sin(y)        # 文字式 sin(y) を変数 g に入れる
```

関数や特定の変数に対する値を求めるには次のようにします。

```
sage: f
x^2
sage: f(5)        # f(5) の値を計算
25
sage: g(1.2)      # g(1.2) を計算
0.932039085967226
```

次のように文字式を定義する方法もあります：

文字式としての関数の定義 2

```
sage: f(x) = x^2          # 変数の指定 (x) があるのが上と異なる
sage: f
x |--> x^2
sage: f(5)
25
```

文字式としての関数の定義が 2 種類あり混乱するかもしれませんが，次の 2 変数関数の扱いを見ると必要性を感じるかもしれません。

2 変数の関数の定義

```
sage: var('y')              # y を文字式として取り扱う宣言
sage: g(x,y) = (x+y)^2      # 関数 g の定義
sage: g
(x, y) --> (x + y)^2
sage: g(3,6)          # x=3, y=6 のときの g の値
81                    # (3+6)^2
sage: g(y=3)          # y=3 のときの g
(x + 3)^2
```

2 変数以上の文字式で特定の変数に数値を入れるには次のようにします：

```
sage: var('y')
sage: h = (x+y)^2
sage: h.subs(y=3)
(x + 3)^2
```

23.2 Sage に用意されている関数

Sage ではじめから様々な初等関数・特殊関数が定義されています。Sage で使える初等関数にはその逆関数 sin, cos, tan, csc, sec, cot ,arcsin, arccos，対数関数・指数関数 log，ln，exp, 双曲関数 sinh，cosh，tahn やその逆関数 arcsinh があります。また代表的な特殊関数であるガンマ関数 gamma(z) ゼータ関数 zeta や楕円関数，いくつかの直交多項式が使えます。これら以外にも数多くの関数が用意されています。

23.3 文字式の展開・因数分解・単純化

関数の展開 (expand)

```
f.expand()
```

または expand(f)。

例えば $(x+1)^3$ を展開するには次のようにします。

```
sage: f = (x+1)^3
sage: f.expand()
x^3 + 3*x^2 + 3*x + 1
```

関数 f の因数分解

```
f.factor()
```

または factor(f)。

$x^3+3x-2x-6$ の因数分解をしてみましょう。

```
sage: f = x^3 + 3*x^2 - 2*x - 6
sage: factor(f)
(x + 3)*(x^2 - 2)
```

factor では整数係数の範囲でしか因数分解されないので，より一般的に多項式の根を求めるには後述の solve を使います。

方程式を単純化することもできます。

関数 f の単純化 (simplify と simplify_full)

```
f.simplify()
```

または simplify(f)。時間はかかるができる限り単純にするには

```
f.simplify_full()
```

または simplify_full(f) とする。

$x^2+2x+5-3x$ を単純化してみましょう：

```
sage: f = x^2 + 2*x + 6 - 3*x
sage: simplify(f)
x^2 - x + 6
```

simplify は加法・減法でできる程度の単純化しかできませんが，simplify_full は三角法による単純化もできます：

```
sage: f = sin(x)^2 + cos(x)^2
sage: f.simplify()       # simplify は sin^2+cos^2=1 を知らない
sin(x)^2 + cos(x)^2      # 変化なし
sage: f.simplify_full() # こちらなら
1                        # 1になる
```

23.4 $x^{105}-1$ の因数分解

計算機により手計算では分からない意外な発見をすることがあるかもしれません。$x^{20}-1$ を整数係数の範囲で因数分解してみましょう。

```
factor(x^20 - 1)
```

答えは

$$(x^8-x^6+x^4-x^2+1)(x^4+x^3+x^2+x+1)(x^4-x^3+x^2-x+1)(x^2+1)(x+1)(x-1)$$

となります。このように，x^n-1 を整数係数で因数分解したとき，係数は ±1 しか出てこないでしょうか？答えは否定的です。次の例を見てみましょう。

```
factor(x^105 - 1)
```

$$
\begin{aligned}
&\left(x^{48}+x^{47}+x^{46}-x^{43}-x^{42}-2\,x^{41}-x^{40}-x^{39}+x^{36}+x^{35}+x^{34}+x^{33}+x^{32}+x^{31}-x^{28}-x^{26}\right.\\
&\quad\left.-x^{24}-x^{22}-x^{20}+x^{17}+x^{16}+x^{15}+x^{14}+x^{13}+x^{12}-x^{9}-x^{8}-2\,x^{7}-x^{6}-x^{5}+x^{2}+x+1\right)\\
&\times\left(x^{24}-x^{23}+x^{19}-x^{18}+x^{17}-x^{16}+x^{14}-x^{13}+x^{12}-x^{11}+x^{10}-x^{8}+x^{7}-x^{6}+x^{5}-x+1\right)\\
&\times\left(x^{12}-x^{11}+x^{9}-x^{8}+x^{6}-x^{4}+x^{3}-x+1\right)\left(x^{8}-x^{7}+x^{5}-x^{4}+x^{3}-x+1\right)\\
&\times\left(x^{6}+x^{5}+x^{4}+x^{3}+x^{2}+x+1\right)\left(x^{4}+x^{3}+x^{2}+x+1\right)\left(x^{2}+x+1\right)\left(x-1\right)
\end{aligned}
$$

となります。赤色の文字で示したように，因数分解したとき ±1 以外の係数 2 が現れます。

23.5　有理式の部分分数展開

有理式を

$$\frac{x^2}{(x+1)^2}=\frac{-2}{x+1}+\frac{1}{(x+1)^2}+1$$

のように分解することを部分分数分解といいます。

有理式の部分分数分解

```
f.partial_fraction()
```

例：

```
sage: f = x^2/(x+1)^2
sage: f.partial_fraction()
-2/(x + 1) + 1/(x + 1)^2 +1
```

元に戻すには `factor` を使います。

```
sage: g = -2/(x + 1) + 1/(x + 1)^2 +1
sage: g.factor()
x^2/(x + 1)^2
```

23.6　連分数

実数 x に対して x を越えない最大の整数を $a_0=\lfloor x\rfloor$ とし，$x_1=1/(x-a_0)$ とおきます。このとき

$$x=a_0+\frac{1}{x_1}$$

です。同様に $a_1=\lfloor x_1\rfloor$，$x_2=1/(x_1-a_1)$ とすると

$$x=a_0+\cfrac{1}{a_1+\cfrac{1}{x_2}}$$

となります。これを繰り返すと

$$x=a_0+\cfrac{1}{a_1+\cfrac{1}{a_2+\cfrac{1}{a_3+\cfrac{1}{a_4+\cdots}}}}$$

という表示が得られます。x の連分数表示といいます。この連分数を簡略化して次の記号で書きます：

$$x = [a_0 : a_1, a_2, a_3, a_4, \cdots].$$

x が有理数ならば連分数表示は有限の長さになることが知られています。例えば

$$\frac{345}{29} = 11 + \frac{26}{29} = 11 + \frac{1}{1 + \frac{3}{26}} = 11 + \frac{1}{1 + \frac{1}{8 + \frac{2}{3}}}$$

$$= 11 + \frac{1}{1 + \frac{1}{8 + \frac{1}{1 + \frac{1}{2}}}}$$

したがって $345/29 = [11; 1, 8, 1, 2]$。連分数表示の重要な特徴は，無理数であっても非常にきれいなパターンで表せることです。

数の連分数表示

実数 a の連分数表示は

```
continued_fraction(a)      # 連分数を返す
continued_fraction_list(a, オプション)    # 連分数のリストを返す
```

オプションは連分数の精度 bits=40 などを指定。

例：

```
sage: continued_fraction(345/29)
[11, 1, 8 , 1, 2]
sage: continued_fraction(sqrt(2))
[1; 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, ...]
```

上の計算では有限個しか表示されていないが，実際は無限の長さの連分数であり $\sqrt{2} = [1; 2, 2, 2, 2, \cdots]$ です。また continued_fraction_list を使うと精度を指定して連分数展開のリストを得ることができます。

```
sage: a = continued_fraction_list(sqrt(3), bits=40)  #40ビットの精度
sage: a
[1, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 3]
```

超越数であっても美しい連分数展開ができる場合があります。自然対数の底の連分数表示は

```
sage: continued_fraction(RealField(400)(e))  # ネピア数の連分数表示
[2; 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10, 1, 1,
12, 1, 1, 14, 1, 1, 16, 1, 1, 18, 1, 1, 20, 1, 1, 22,
 1, 1, 24, 1, 1, 26, 1, 1, 28, 1, 1, 30, 1, 1, 32, 1,
1, 34, 1, 1, 36, 1, 1, 38, 1, 1, 40, 1, 1, 42, 1, 1,
44, 1, 1, 46, 1, 1, 48, 1, 1, 50, 1, 1, 52, 1, 1, 54,
1, 1, 56, 1, 1, 58, 1, 1, 60, 1, 1, 62, 1, 1, 64, 1, 1,
66, 1, 1, 68, 2]
```

となります。上で RealField(400)(e) としたのは e の近似値を使うためです。

23.7 極限と条件

Sage では無限大 ∞ を oo のように小文字のオーを二つ並べて表します。関数の極限 $\lim_{x\to a} f(x)$ を求めるには limit(....) を用います。

極限の計算

$$\lim_{x\to a} f(x) \qquad \lim_{x\to a+0} f(x) \qquad \lim_{x\to a-0} f(x)$$

はそれぞれ次のように計算します。

```
f.limit(x = a)            # または limit(f, x=a)
f.limit(x=a, dir='+')     # 右極限
f.limit(x=a, dir='-')     # 左極限
```

試しに次式の右辺の極限を計算してみましょう：

$$e = \lim_{x\to\infty} \left(1+\frac{1}{x}\right)^x$$

```
sage: f = (1 + 1/x)^x
sage: f.limit(x=oo)
e    # ネピア数
sage: n(e)
2.71828182845905
```

右極限と左極限を計算してみます。

```
sage: limit(1/x, x=0, dir='+')    # 右極限
+Infinity                         # +∞
sage: limit(1/x, x=0, dir='-')    # 左極限
-Infinity                         # -∞
```

そして左極限と右極限が一致しないのですが

```
sage: limit(1/x, x=0)
Infinity
```

となるので，Infinity は必ずしも $+\infty$ を意味していないことがわかります。

さて次に極限

$$\lim_{x\to 0} x^a \tag{6}$$

を求めることを考えてみます。これが定まるかどうかは a の値に依存します。このようなとき assume によって a の性質を指定すると解決する場合があります。

以下の命令を，一回実行するごとに Restart worksheet を繰り返しながら実行してみよう：

```
var('a')
limit(x^a, x=0)
```

実行結果の例

```
Traceback (click to the left of this block for traceback)
...
Is  a  positive, negative, or zero?       # エラーメッセージの最後
```

そこで $a > 0$ と仮定します

```
var('a')
assume(a>0)      # a>0と仮定する
limit(x^a, x=0)
```

実行結果の例

```
Traceback (click to the left of this block for traceback)
...
Is a an integer?            # まだ値が定まらない。
```

そして右極限を指定すると求まります：

```
var('a');  assume(a>0)
limit(x^a, x=0, dir='+')      # 右極限
```

実行結果の例

```
0
```

また a が偶数であると指定しても極限は求まります：

```
var('a')
assume(a,'even')
limit(x^a, x=0)
```

実行結果の例

```
0
```

23.8 代数的な和の計算

Sage で和を計算するには sum を使います。sum 関数は Python でも用意されていました：

```
>>> sum([1,2,3,4,5,6,7])
28
```

Sage でも同じ関数名で和を求めることができますが，有限和だけでなく無限級数も厳密に計算することもできます。

$\sum_{k=a}^{b} f(k)$ を計算する

```
var('k')              # kを変数とする
sum(f(k),k,a,b)
```

1 行目のように和をとるときのダミー変数 k を定義することに注意。

$1+2+\cdots+50$ を求める：

```
sage: var('k')
sage: sum(k,k,1,50)
1275
```

sum は数値としての和の算出だけでなく

$$\sum_{k=1}^{m} k^2 = \frac{m(m+1)(2m+1)}{6}$$

のように厳密に成り立つ式も導出してくれます：

```
sage: var('k m')                    # kとmを変数とする
sage: ans = sum(k^2,k,1,m)          # 和を計算してansに代入
sage: ans
1/2*m^2 + 1/2*m                     # これが答え
sage: factor(ans)                   # ansを因数分解
1/6*(2*m + 1)*(m + 1)*m             # 上の答えを因数分解したもの
```

次のような公式を得ることも出来ます：

$$\sum_{k=0}^{\infty} \frac{1}{k!} = e, \qquad \sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}, \qquad \sum_{k=1}^{\infty} \frac{2^{-k}}{k(k+1)} = -\log 2 + 1$$

```
var('k')
print(sum(1/factorial(k),k,0,oo))
print(sum(1/k^2,k,1,oo))
print(sum(2^(-k)/(k*(k+1)),k,1,oo))
```

実行結果の例

```
e
1/6*pi^2
-log(2) + 1
```

24　グラフの描画とデータの可視化

データの可視化はコンピューターの最も重要な活用の一つです。Sage では手軽にさまざまなデータの描画を行うことができます。以下では，Sage を jupyter ノートブック内で実行していると仮定して解説します。

24.1　関数のグラフを描画（plot）

関数 $f(x)$ のグラフをプロットするには『plot』を使います。

plot——関数 $f(x)$ のグラフをプロット

```
plot(f(x), a,b)
plot(f(x), x, a, b)     # これでもよい
plot(f(x), (x,a,b))     # これでもよい
plot(f(x), (a,b))       # これでもよい
```

- 関数 $f(x)$ を x の範囲 $[a,b]$ で描画する。

試しに $\sin(x)$ を区間 $[-6,6]$ でプロットしてみましょう。

例 1 ： 1
```
plot(sin(x), -6,6)
```

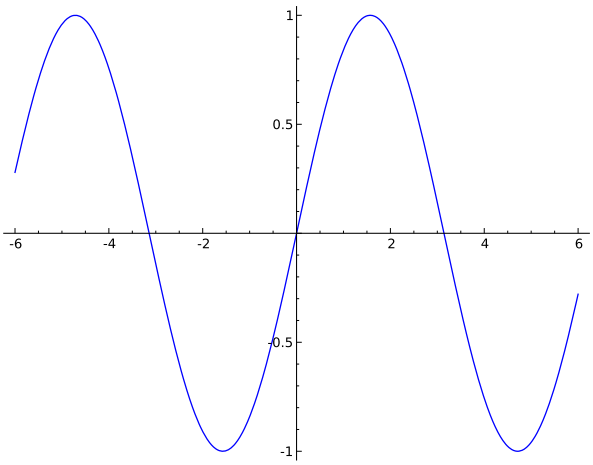


図 8　出力結果：$\sin(x)$ のグラフ

変数 y の関数の場合は次のようにします (グラフは上と同じ)。

```
1 var('y')
2 plot(sin(y),y,-6,6)
```

つぎに原点で発散している関数 $1/x$ のグラフを描いてみましょう：

例 **2**： 1
```
plot(1/x, -2,2)
```

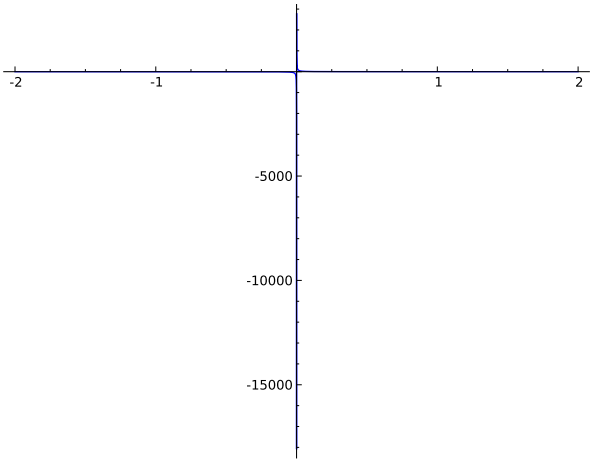


図 9　出力結果：$1/x$ のグラフ

グラフの縦軸が極端に大きい数値になったためにグラフがつぶれてしまいました。これは $1/x$ のグラフが原点で発散しているためです。このような関数を描画するには関数の値の範囲—**値域**—を指定しないといけません：

関数 $f(x)$ のグラフを値域を指定して描画

```
1 plot(f(x), a,b, ymin=c, ymax=d)
2 plot(f(x), (x,a,b), ymin=c, ymax=d)    # こちらでもよい
```

- 関数 $f(x)$ を x の範囲を $[a,b]$ として描画。
- y 軸の最小値・最大値はそれぞれ `ymin=c, ymax=d`。

関数 $1/x$ の値域を $[-3,4]$ に制限してプロットします。

例３：
```
1 plot(1/x, -2, 2, ymin=-3, ymax=4)
```

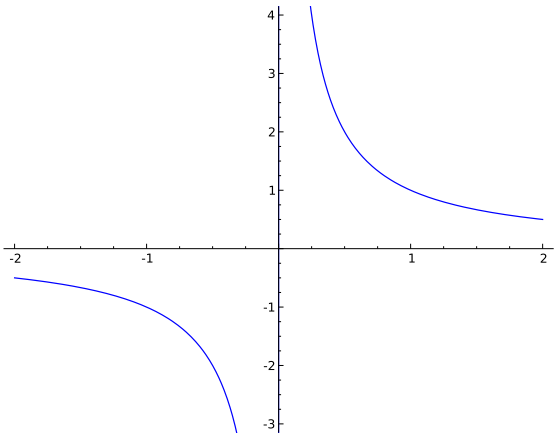


図 10　出力結果：$1/x$ を値域 $[-3,4]$ に制限したグラフ

複数の関数のグラフを一度にプロットすることもできます。例えば 3 つの関数 $\sin(x), \cos(x), \tan(x)$ を区間 $[-6,6]$，値域 $[-5,5]$ で重ねて描画するには次のようにします：

例４：
```
1 plot( (sin(x),cos(x),tan(x)), -6,6, ymin=-5,ymax=5)
```

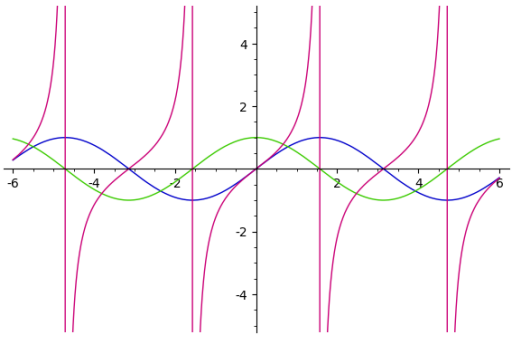


図 11　出力結果：$\sin(x), \cos(x), \tan(x)$ のグラフ

plot のオプション

plot では，オプションを指定することにより色を付けたり，グラフや座標軸に名前を付けたりすることが出来ます。書式は次の通りです

```
1 plot( f(x), (x,a,b), color='グラフの色',
2       axes_labels=['横軸名','縦軸名'], legend_label='グラフ名')
```

- 『色』は red，yellow，blue，green や RGB カラー'#3F4A46' などを指定。
- axes_labels で軸のラベルを記述。
- 座標軸を消すには axes=False を追加
- ラベル名ではドル記号$ ··· $で囲むことで簡単な LaTeX の数式環境が使えます。そこでは\sin(x) で $\sin(x)$，\\tan(x) で $\tan(x)$，\\frac{a}{b}で $\frac{a}{b}$ を表します。

これらのオプションは例えば次のように使用します：

例５：
```
1 plot(sin(x^2),(x,0,5), color='red', axes_labels=['time','amplitude'],
2      legend_label='$\sin(x^2)$')
```

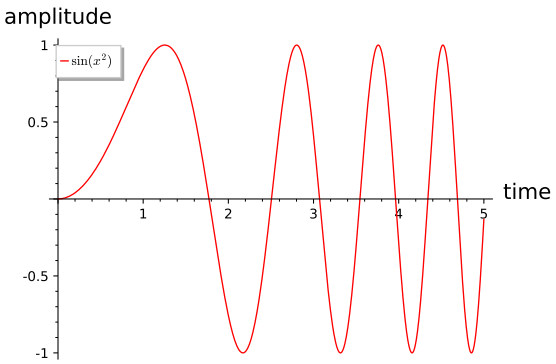


図 12　出力結果：$\sin(x^2)$ のオプション付きプロット

24.2　グラフィックスで使える色

red, blue 意外にも様々な色が用意されています。使える色は colors という辞書に記録されています：

```
1 for i in sorted(colors):
2     print(i)
```

実行結果の例

```
aliceblue
antiquewhite
aqua
...
...
...
yellow
yellowgreen
```

24.3　グラフ描画：応用編

グラフなどのオブジェクトに対して，それを画面上に描画する命令 show() について解説します。

show の使い方 1

```
1 p1 = plot( sin(x), -6,6 )
2 p2 = plot( tan(x), -6,6 )
3 p1.show()
4 p2.show(ymin=-5, ymax=5)
```

- 1 行目で $\sin(x)$ のグラフデータを変数 p1 に代入。3 行目で p1 を画面に表示させています。
- 4 行目で p2 の値域を制限して描画しています。

上では 3 行目は show(p1) でも同じです。次の例のように，グラフを重ねて表示することができます：

例 6：

```
1 p1 = plot(sin(x),-5,5, legend_label='sin', color="red")
2 p2 = plot(cos(x),-5,5, legend_label='cos', color="blue")
3 p3 = plot(tan(x),-5,5, legend_label='$\\tan(x)$', color="green")
4 p4 = p1+p2+p3                #p4はグラフp1,p2,p3を重ねたもの
5 p4.show(ymin=-3,ymax=3)      #p4を表示させるにはshowを用いる
```

上のプログラムでは $\sin, \cos, \tan$ のグラフのグラフィックが個別に作れれて変数 p1,p2,p3 に入れられ，4 行目で重ねられたグラフ p4 が作られます。5 行目で p4 が描画されます。

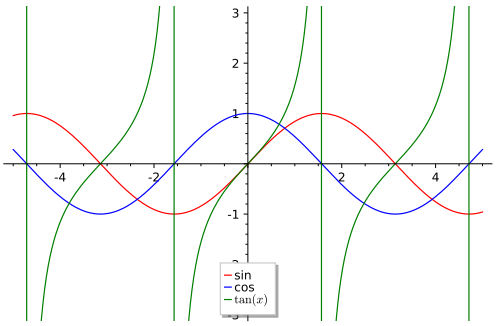


図 13　出力結果：$\sin(x), \cos(x), \tan(x)$ の色を変えて重ねて描画

グラフを並べて描画するには graphics_array を使います：

二つのグラフを並べて表示

例７：

```
1 p1 = plot( sin(x), -4,4, aspect_ratio=2)
2 p2 = plot( cos(x), -4,4, aspect_ratio=2)
3 p3 = graphics_array([p1,p2])
4 p3.show()
```

ここで aspect ratio とは縦横比のことです。

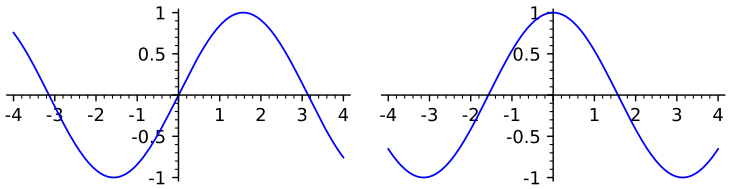


図 14　出力結果：$\sin(x), \cos(x)$ のグラフを横に並べる

グラフの上下やグラフによって囲まれた領域を塗りつぶすことが出来ます：

グラフの塗りつぶし (filling)

例８：

```
1 p1 = plot(sin(x), (x,-5,5), fill = 'axis') # 横軸との間を塗りつぶし
2 p2 = plot(sin(x), (x,-5,5), fill = 'min') # グラフの下部を塗りつぶし
3 p3 = plot(sin(x), (x,-5,5), fill = 'max') # グラフの上部を塗りつぶし
4 p4 = plot(sin(x), (x,-5,5), fill = 0.5) #0.5との間を塗りつぶし
5 graphics_array([[p1, p2], [p3, p4]]).show()
```

fill オプションによって塗りつぶす領域を指定します。

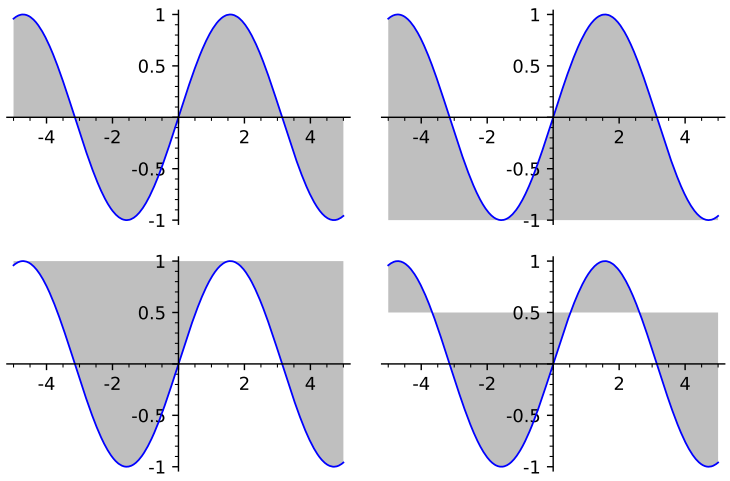


図 15　出力結果：fill の使い方

上のプログラムでは，`graphics_array([[a,b],[c,d]])` によってグラフを $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ の順に並べています。

2 つの関数の間を塗りつぶすこともできます。例えば $\sin(x)$ と x^2-1 との間を塗りつぶすには次のようにします。

例 9：

```
plot( sin(x), (x,-2,2), fill = x^2-1)
```

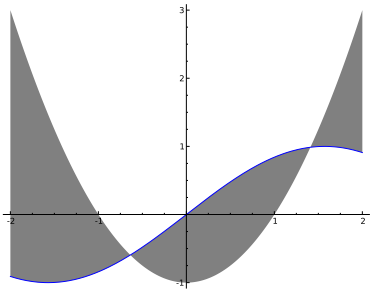


図 16　出力結果：$\sin(x)$ のグラフ，x^2-1 との間を色づけしている

他にもグラフの線の太さ (thickness) を変えたり，線を点線 (dashed) にする等のさまざまなオプションがあります。これらのオプションを参照するには，plot のヘルプをみてください：

plot のヘルプを表示

```
plot?
```

24.4　陰関数のグラフ

$f(x,y)=0$ で定義される x,y 平面の曲線を描くには `implicit_plot` を使います。

2 変数の陰関数のグラフを描く

```
var('y')
implicit_plot(f(x,y), (x,a,b), (y,c,d))
```

- 2 変数関数 $f(x,y)=0$ のグラフを $(x,y)\in[a,b]\times[c,d]$ の範囲で描画する。

ここで `var('y')` は `y` をもう一つの変数として取り扱うために必要な宣言です。たとえばデカルトの正葉線 (x^3+y^3-3xy) を描くには次のようにします：

例 **10**：

```
var('y')
implicit_plot(x^3+y^3-3*x*y, (x,-2,2), (y,-2,2))
```

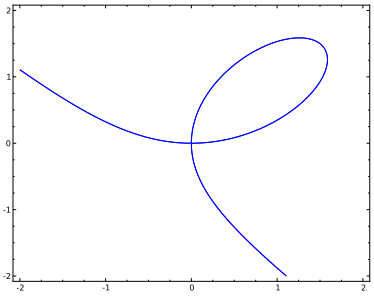


図 17　出力結果：デカルトの正葉線

座標が $(x(t), y(t))$ のように一つのパラメーターに依存して動く点の軌跡を描画するには `parametric_plot` を用います：

$(x(t), y(t))$ で定義される軌跡の描画

```
1 var('t')
2 parametric_plot([cos(t) + 2*cos(t/4), sin(t)-2*sin(t/4)],
3       (t,0, 8*pi), fill=true)
```

例 11：
上の例では $x(t) = \cos(t) + 2\cos(t/4), y(t) = \sin(t) - 2\sin(t/4)$ です。パラメーター t の動く範囲は 8π にしています。またオプション `fill` によって，囲まれる領域を色づけしています：

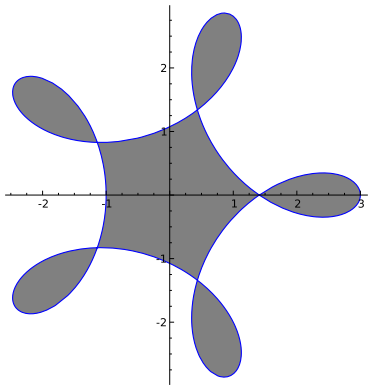


図 18　出力結果：parametric plot(with filling)

24.5　リストのプロット

リスト化されたデータをプロットするには `list_plot` を使います：

リストのプロット

```
a = リスト
list_plot(a)
```

- プロットできるリストは数値のデータのリスト [3,2,6,3,12,-2,2] や
- 2 次元ベクトルからなる [(1,2),(4,1),(3,4),(4,2)] のようなリストです。
- 3 次元のデータのからなるリストに対しては空間的なプロットがなされます。

例 12：

```
a = [1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9,10,10]
list_plot(a)
```

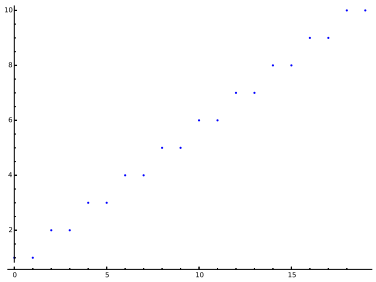


図 19　出力結果：リストのプロット

リストの最初は 0 番目，次は 1 番目となることに注意してください。上と同じリストでプロットのオプションに plotjoined=True を指定すると点同士が直線でつながります。

例 13：

```
a = [1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9,10,10]
list_plot(a, plotjoined=True)
```

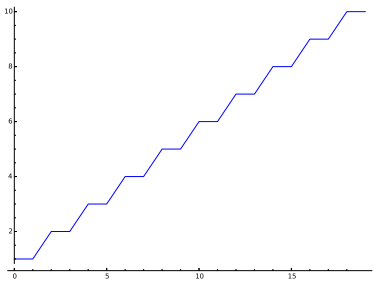


図 20　出力結果：リストのプロット (plotjoined=True)

例えば 2 次元の点データのプロットは次のようにします：

例 14：

```
a = [(sqrt(j)*cos(j),sqrt(j)*sin(j)) for j in range(20)]x
list_plot(a,plotjoined=True)
```

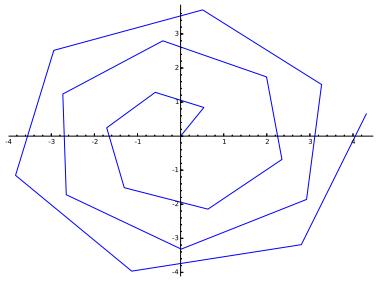


図 21　出力結果：$\{\sqrt{j}(\cos(j), \sin(j))|j = 0, 1, \cdots, 19\}$ をプロット

24.6　変数 (x, y) の取り得る 2 次元領域を描く

関数 $f(x, y)$ が与えられたときに $f(x, y) > 0$ となる領域を描くには `regin_plot` を用います：

region_plot：$f(x, y) > 0$ となる (x, y) の領域を描く

```
var('y')
region_plot(f(x,y)>0, (x,a,b), (y,c,d))
```

たとえば $\sin(x^2 - y^3) > 0$ となる x, y の領域を描くには次のようにします。

例 **14**：

```
var('y')
region_plot(sin(x^2-y^3)>0, (x,-3, 3), (y,-3, 3))
```

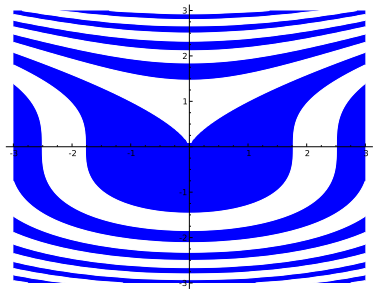


図 22　出力結果：$\sin(x^2 - y^3) > 0$

変数 (x, y) に対する条件は `[f(x,y)>0,g(x,y)>0,h(x,y)>0]` のようにリストにすることにより複数指定することが出来ます。

例 **15**：

```
y = var('y')
region_plot([x>0, y>0, x^2+y^2>0.5], (x,-1,1), (y,-1,1) )
```

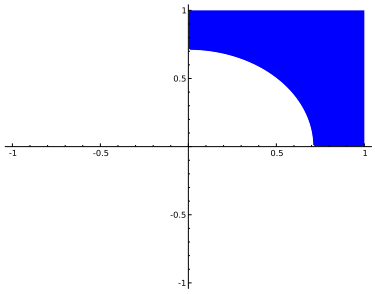


図 23　出力結果：$x > 0, y > 0, x^2 + y^2 > 0.5$ となる x, y の領域のグラフ

region_plot は指定された定義域を 100 等分して条件をチェックして描画を行っています。変化の激しい関数の場合は，条件が十分正しくチェックされずに大幅に間違ったグラフが出力される可能性があります。このときは plot_points=300 のようにオプションを指定することにより，プロットする点の数を増やしましょう。

24.7　基本的なパーツ (円・楕円・矢印・円弧・線分・点・テキスト)

Sage の 2 次元グラフィックスでは，円や線分などの基本的なパーツを手軽に描くことができます。これらのパーツは plot などと組み合わせて使うことができます。

次が代表的なオブジェクトです：

- `circle((a,b),r)`：中心 (a, b), 半径 r の円周
- `ellipse((a,b),c,d)`：中心 (a, b)，横の半径 c, 縦の半径 d の楕円
- `arrow((a,b),(c,d))`：始点 (a, b), 終点 (c, d) の矢印
- `arc((a,b),r,sector=(c,d))`：中心 (a, b), 半径 r, 角度 (c, d)。角度 (c, d) は，たとえば 0 度から 90 度までの円弧なら $(0, \pi/2)$ のようにラジアンで表します。
- `line([(a,b),(c,d)])`：(a, b) と (c, d) を結ぶ線分。オプションとして `linestype='--'` を指定すると点線となります。また `marker='o'` のオプションを付けると線分の両端に点がつきます。
- `point(a,b,size=r)`：位置 (a, b)，大きさ r の点
- `text('文字', (a,b), fontsize=r)`：位置 (a, b) にあるサイズ r の文字

例 **16**：
```
1 s1 = plot(x^2,(x,-1,1))                                  # 放物線
2 s2 = point((0,0), size=100, color='black')               # 点
3 s3 = arrow((-1/2,1/4),(1,1), color='red')                # 矢印
4 s4 = arc((0,0),0.5,sector=(0,pi), color='green')         # 円弧
5 s5 = s1 + s2 + s3 + s4
6 s5.show()
```

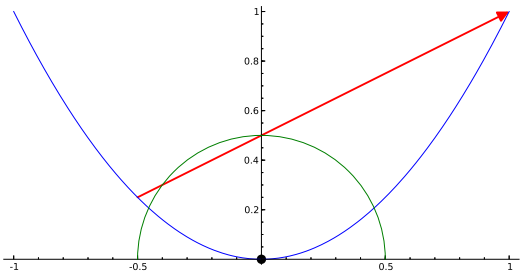


図 24　出力結果：グラフと点と矢印と円弧

25　2 変数関数の可視化

ここでは実数値の 2 変数関数 $f(x,y)$ の可視化を解説します。

25.1　3 次元のプロット

2 変数の実数値関数 $f(x,y)$ を 3 次元的にプロットするには plot3d を使います。

plot3d—$f(x,y)$ の 3 次元的な描画

```
var('y')
plot3d(f(x,y), (x,a,b),(y,c,d))
```

次のようなオプションがあります。

- plot_points：サンプルする点の数 (多くすればするほど精密なグラフになる)
- opacity：透明度 (0 から 1 の値，0.5 なら半透明になる)
- aspect_ratio：縦横高さの比（[1,2,1] のように比を指定する）

例 17：
```
var('y')
plot3d(sin(x-y)*y*cos(x), (x,-3,3), (y,-3,3),
     opacity=0.5, aspect_ratio=[1,1,1])
```

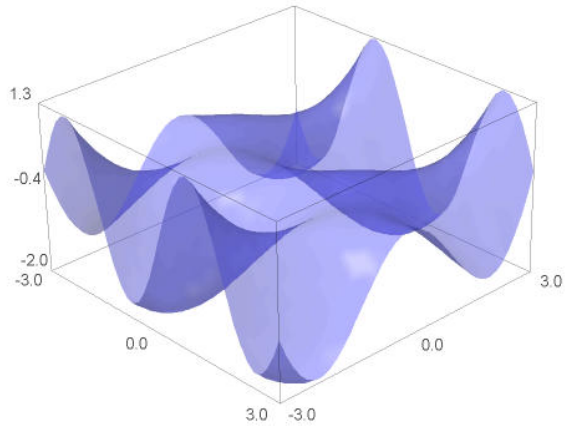


図 25　出力結果：$y\sin(x-y)\cos(x)$ のグラフ

25.2　密度プロット（density plot）と等高線プロット（contour plot）

plot3d の他にも密度プロットは関数の高さを色の濃さであらわす density plot や，等高線であらわす contour plot があります。密度プロットでは高い方が明るく，低い方が濃くなります。

例 18：
```
var('y')
density_plot(sin(x*y), (x,-3,3), (y,-3,3))
```

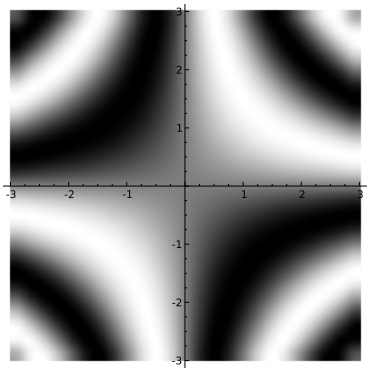


図 26　出力結果：$\sin(xy)$ の密度プロット

例 **19**：
```
x, y = var('x y')
contour_plot(cos(x)+cos(y), (x, -8, 8), (y, -8, 8),contours=20)
```

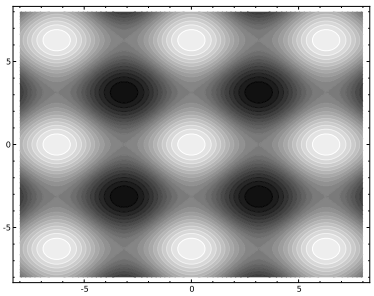


図 27　出力結果：$\cos(x)+\cos(y)$ の等高線プロット

25.3　その他の 3 次元プロット

上記以外の 3 次元描画には次のようなものがあります。

- `implicit_plot3d`
- `parametric_plot3d`
- `plot_vector_field3d`

25.4　画像の保存

描いた画像を保存する方法はいくつかありますが Jupyter ノートブックを使っていて png 形式で保存する場合は，出力された画像を右クリックして『名前を付けて保存』から保存するだけでできます。png 形式以外では次のようにすることで pdf, eps, ps 等の形式で保存することができます：

notebook での画像の保存の仕方

```
p1 = plot(sin(x), (x,-4,4))
p1.save("ファイル名.pdf")
```

- 上を実行すると画像が表示される代わりに画像がフォルダに生成されます。CoCalc であれば，左上の **Files** から生成された画像をダウンロードすることができます。自分の PC から Jupyter notebook を起動している状態であれば notebook ファイルがおいてあるディレクトリもしくはホームディレクトリに画像が生成されます。
- eps で保存するなら上で pdf の部分を eps に変えます。
- 3 次元の絵は eps, pdf, ps などの形式では保存することができませんが，png 形式で保存することができます。

Sage のプログラム (**.sage 形式のファイル) が端末から実行できる環境であれば，直接ファイルを保存することもできます。これは大量の画像を生成するときに特に便利です。

- ファイル名：**plot.sage**

```
p1 = plot(sin(x), (x,-4,4))
p1.save("plot.pdf")      # p1をpdfファイルとしてセーブする
```

上のファイルを Emacs などで作成したら，端末から `sage plot.sage` と実行することにより，ディレクトリに直接画像のファイルが生成されます[*15]。

26　グラフの描画に関するおもしろい例題

複素関数 e^z は零点を持ちませんが，マクローリン展開を有限項で切った多項式

$$\sum_{k=0}^{n} \frac{z^k}{k!}$$

は n 個の零点を持ちます。不思議なことに関数を $z \to nz$ とスケーリングすると

$$\sum_{k=0}^{n} \frac{(nz)^n}{k!} \tag{7}$$

の零点の集合は $n \to \infty$ の極限で関数 $|ze^{z-1}| = 1$ の滴の部分に一致します（この事はすでに証明されていますが，どうやって証明するんでしょう？）。数値計算でこれを試してみたいと思います。

多項式の零点を求めるには `roots` を使います。例えば $x^5 + 3x + 1 == 0$ の零点を求めるには

```
eq = x^5+3*x+1==0
eq.roots(x, ring=CC, multiplicities=False)
```

とします。実行結果は

実行結果の例

```
[-0.331989029584509, -0.839072433066608 - 0.943851550132862*I,
-0.839072433066608 + 0.943851550132862*I, 1.00506694785886 -
0.937259156692892*I, 1.00506694785886 + 0.937259156692892*I]
```

となります。`multiplicities` は多項式の根の多重度を表示させるかどうかのオプションです。いまは必要ないので `False` にしています。

さて，7 の零点を求めてみましょう：

[*15] CoCalc などの環境からはもちろんできません

```
var('z k')    #z, kを変数とする
nn = 30        #nnの値を30とする
f = sum((nn*z)^k/factorial(k), k,0,nn) # 関数fを定義
eq = f==0                              # 方程式f==0をeqと名付ける
lis = eq.roots(z, ring=CC, multiplicities=False);    # eqの根の集合をlisとする
list_plot(lis)      #lisをプロットする
```

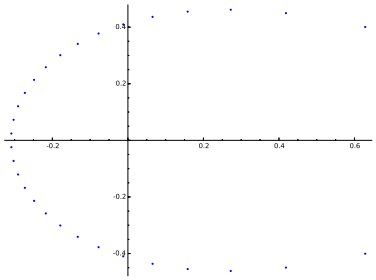


図 28　出力結果：$n = 30$ のときの零点の集合

点の数を増やしてみましょう。

```
var('z k')    #z, kを変数とする
nn = 200       #nnの値を200とする
f = sum((nn*z)^k/factorial(k), k,0,nn) # 関数fを定義
eq = f==0                              # 方程式f==0をeqと名付ける
lis = eq.roots(z, ring=CC, multiplicities=False);    # eqの根の集合をlisとする
list_plot(lis)      #lisをプロットする
```

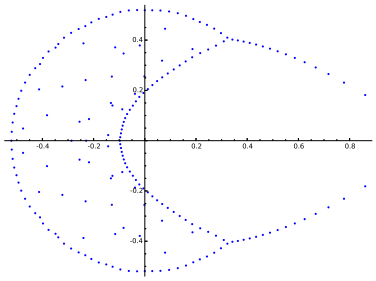


図 29　出力結果：$n = 100$ のときの零点の集合?（誤差で変な結果になる）

この結果は信用できないので，roots のオプションで計算精度を高くしてみたいと思います。そのためには CC で計算していたところを ComplexField(300) のように精度を上げて計算します。

```
var('z k')
nn = 200        # nnの値を200とする
f = sum((nn*z)^k/factorial(k), k,0,nn)
eq = f==0
lis = eq.roots(z, ring=ComplexField(300), multiplicities=False) # 300bitで計算
list_plot(lis)
```

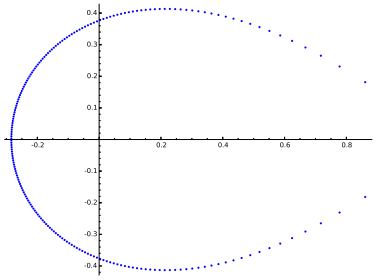


図 30　出力結果：$n = 200$ のときの零点の集合

次に $|ze^{1-z}| = 1$ を平面上に描いてみましょう。$z = x + iy$ のとき $|ze^{1-z}| = \sqrt{x^2+y^2}e^{1-x}$ なので

```
var('x y')
implicit_plot(sqrt(x^2+y^2)*exp(1-x)-1, (x,-1,2), (y,-1,1),color='red')
```

とすればプロットすることができます。これを上で求めた零点と重ねてみましょう。

```
var('z k')
nn =200
f = sum((nn*z)^k/factorial(k), k,0,nn)    # 関数 f を定義
eq = f==0
lis = eq.roots(z, ring=ComplexField(300), multiplicities=False);
p1 = list_plot(lis)
var('x y')
p2 = implicit_plot(sqrt(x^2+y^2)*exp(1-x)-1, (x,-1,2), (y,-1,1),color='red')
(p1+p2).show()
```

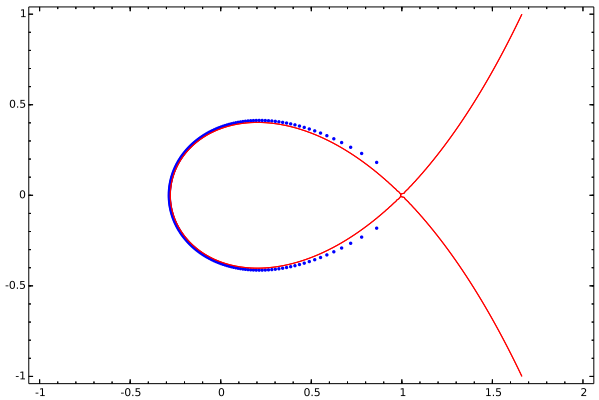


図 31　出力結果：零点の集合と $|ze^{1-z}| = 1$ のグラフを重ねる

青い点（ゼロ点）と赤い線の滴の部分はよく一致していることが見て取れます。

複素関数を色を付けて表示する complex_plot というコマンドがあります，これを使って $\sum_{k=1}^{40}(40z)^k/k!$ を描くと次のようになります。

```
var('k')
nn=40
f = sum((nn*x)^k/factorial(k),k,0,nn)
complex_plot(f,(-1,2),(-1,1),plot_points=500,aspect_ratio=1)
```

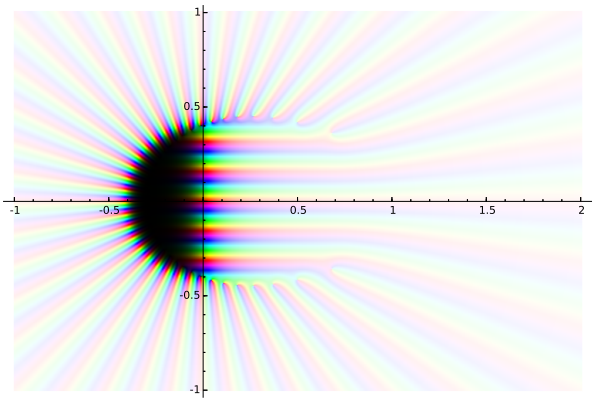


図 32　出力結果：複素関数 $\sum_{k=1}^{40}(40z)^k/k!$ の形

27　練習問題

ここまでの Sage の解説をすべて Jupyter ノートブックで実行し，その Jupyter ノートブックファイルを保存して提出すること。ファイル名は `sage01.ipynb` とすること。念のため，ファイルを保存したら，自分でファイルの読み込み（や Upload(CoCalc の場合)）を行ってみて，正常に読み込めるかどうか確認しましょう。

28　方程式の扱い

28.1　方程式を意味する等号 “==”

Python では『=』は右辺を左辺に代入することを意味しました。例えば

```
>>> x = 3
>>> print(x)
3
```

と書けば変数 x が新たに作られ，3 が代入されます。また，『==』は方程式の真偽値を返し，if 文や while 文などの条件を記述するときに用いました。たとえば

```
>>> 2**3 == 8
True
>>> 8 > 9
False
```

のようになります。Sage ではこれらとは別に『==』には形式的な記号としての等号の意味があります。例えば

```
sage: var('y')
sage: x^2 + y^2 == x*y + 1     # これは単なる方程式
```

と書いても何も返されませんが，上の方程式を一つの記号式として変数に代入することができます：

```
sage: var('y')                          # y も不定元(=形式的文字)とする。
sage: f = x^2 + y^2 == x*y + 1     # 変数fに方程式x^2+y^2==x*y+1を代入
```

このとき，次のように f の右辺や左辺を取り出す事ができます：

```
sage: f              # fは？
x^2 + y^2 == x*y + 1
sage: f.rhs()        #fの右辺は？
x*y + 1
sage: f.lhs()        #fの左辺は？
x^2 + y^2
```

ここで rhs，lhs はそれぞれ右辺（right hand side），左辺（left hand side）の略です。

28.2　方程式の厳密解を求める（solve）

$f(x) = g(x)$ を満たす x を求めるには solve という関数が用意されています：

方程式を解く

```
solve(f(x)==g(x),x)      # f(x)=g(x)をxについて解く
solve([eq1,eq2], x, y)   # 連立方程式eq1, eq2を解く
```

ここで eq1，eq2 はそれぞれ x,y に関する方程式とします。

1 次方程式 $2x + 3 = 7$ を解きます：

```
sage: solve(2*x+3==7,x)
[x == 2]
```

変数 x, y についての連立方程式を解くには，まず変数 y を定義する宣言をします。例えば連立方程式 $x+y=6$, $x-y=4$ は次のように解きます：

```
sage: var('y')
sage: solve([x+y==6, x-y==4], x,y)
[[x == 5, y == 1]]
```

さて 3 次方程式 $x^3+2*x+1==0$ を解いてみましょう：

```
solve(x^3+2*x+1==0, x)
```

3 次方程式の解は 3 つあり，次のようになります：

実行結果の例

```
[x == -1/2*(1/18*sqrt(59)*sqrt(3) - 1/2)^(1/3)*(I*sqrt(3) + 1) - 1/3*(I*sqrt(3) - 1)
/(1/18*sqrt(59)*sqrt(3) - 1/2)^(1/3), x == -1/2*(1/18*sqrt(59)*sqrt(3) - 1/2)^(1/3)
*(-I*sqrt(3) + 1) - 1/3*(-I*sqrt(3) - 1)/(1/18*sqrt(59)*sqrt(3) - 1/2)^(1/3), x
 == (1/18*sqrt(59)*sqrt(3) - 1/2)^(1/3) - 2/3/(1/18*sqrt(59)*sqrt(3) - 1/2)^(1/3)]
```

これは見づらいですが，解はリストの形をしているので，次のように好きな部分を取り出すことができます。

```
a = solve(x^3+2*x+1==0,x)
print(a[0])
print(a[1])
print(a[2])
```

実行結果の例

```
x == -1/2*(1/18*sqrt(59)*sqrt(3) - 1/2)^(1/3)*(I*sqrt(3) + 1) - 1/3*(I*sqrt(3) - 1)/(1/18*sqrt(59)*sqrt(3) -
x == -1/2*(1/18*sqrt(59)*sqrt(3) - 1/2)^(1/3)*(-I*sqrt(3) + 1) - 1/3*(-I*sqrt(3) - 1)/(1/18*sqrt(59)*sqrt(3)
x == (1/18*sqrt(59)*sqrt(3) - 1/2)^(1/3) - 2/3/(1/18*sqrt(59)*sqrt(3) - 1/2)^(1/3)
```

solve では上のような 3 次方程式や 4 次方程式は根号を使って解くことが可能ですが，よく知られているように 5 次以上の方程式は一般的には解くことがきません。そのような場合は入力した方程式がそのまま出力されます：

```
sage: solve(x^5+3*x+1==0,x)
[0 == x^5 + 3*x + 1]
```

もちろん特別な係数をもつ方程式については代数的に解ける場合があります。

```
solve(x^5+x+1==0,x)
```

実行結果の例

```
[x == -1/2*(I*sqrt(3) + 1)*(1/18*sqrt(3)*sqrt(23) - 25/54)^(1/3) -
1/18*(-I*sqrt(3) + 1)/(1/18*sqrt(3)*sqrt(23) - 25/54)^(1/3) + 1/3,
x == -1/2*(-I*sqrt(3) + 1)*(1/18*sqrt(3)*sqrt(23) - 25/54)^(1/3) -
1/18*(I*sqrt(3) + 1)/(1/18*sqrt(3)*sqrt(23) - 25/54)^(1/3) + 1/3,
x == (1/18*sqrt(3)*sqrt(23) - 25/54)^(1/3) + 1/9/(1/18*sqrt(3)*sqrt(23) -
25/54)^(1/3) + 1/3,
x == -1/2*I*sqrt(3) - 1/2, x == 1/2*I*sqrt(3) - 1/2]
```

また三角関数を含む方程式に対しても解を求めることもできるようです：

```
sage: solve(sin(x)==1/2,x)
[x == 1/6*pi]
```

しかし，もう一つの解 $x=\frac{1}{6}\pi-\pi$ は求められていないようです。さらに三角関数の加法定理などを必要とする方程式は解けないようです：

```
sage: solve(sin(x)*cos(x)==1/2,x)
[sin(x) == 1/2/cos(x)]
```

以上のように solve で解ける方程式もありますが解けない方程式も多数あります。

28.3 方程式の数値解（find_root）

方程式の数値解を求めるには find_root を使います：

方程式の数値解を区間 $[a,b]$ の中で探す

```
find_root(方程式,a,b)
```

- 区間の中に解が複数個存在しても最初に見つけた 1 つの解しか返しません。

例：$x^2+2x-9=0$ の解を $[-5,5]$ の中で探すには次のようにします：

```
sage: find_root(x^2+2*x-9==0,-5,5)
2.162277660168379
```

上と同じ方程式で解を探す範囲を狭くすると解が存在しなくなりエラーが返されます。

```
find_root(x^2+2*x-9==0,-2,2)
```

実行結果の例

```
Traceback (click to the left of this block for traceback)
...
RuntimeError: f appears to have no zero on the interval
```

エラーメッセージでは方程式 f はその区間では 0 にならなかったといっています。find_root の利点は，答えは近似値であって数学的には厳密ではないけれども，かなり複雑な方程式に対しても数値解を得ることができる点にあります：

```
sage: find_root(tan(x)==8-x^3,-10,10)
2.1261689044831993
```

解付近の振る舞いが非常に悪い関数関数では，それほど正確な答えが出ないこともあります：

```
sage: find_root(x^400,-5,5)
0.13155619723565876
```

もちろん上の方程式の解は $x=0$ のみのはずです。これは数値解を求めるときに Newton 法という方法を用いているためで，この方法では微分係数が非常に小さくなる点の付近では近似の精度が悪くなります。

29 微分と積分

29.1 微分

関数の微分は diff や derivative で行います。

$f(x)$ の微分

```
sage: diff(sin(x),x)
cos(x)
sage: derivative(tan(x),x)
tan(x)^2 + 1
```

29.2 積分

積分を計算することもできます。

積分の計算

関数 $f(x)$ の不定積分 $\int f(x)dx$ を求める。

```
integral(f(x),x)          # 関数f(x)の不定積分を計算
integral(f(x),x,a,b)    # f(x)の[a,b]上の定積分を計算
```

• integrate でも計算できます。• これらの結果は代数的に計算され結果は厳密です。• 不定積分の結果に積分定数はありません。

例：

```
sage: integral(sin(x),x)
-cos(x)
sage: integral(x^4)
1/5*x^5
sage: integral(1/(1-x^3))
1/3*sqrt(3)*arctan(1/3*(2*x + 1)*sqrt(3)) - 1/3*log(x - 1)
+ 1/6*log(x^2 + x + 1)
```

もちろん，いつでも原始関数が求まるわけではありません：

```
sage: integral(sin(x)/log(x),x)
-(log(x)*integrate(cos(x)/(x*log(x)^2), x) + cos(x))/log(x)
```

上の計算では，integrate が残っており積分が求まったとはいえません。

被積分関数に他の変数が混じっている場合は，次のように変数の宣言をしてから積分の命令を実行します：

```
sage: var('a')
sage: integral(cos(a*x),x)
sin(a*x)/a
```

次に定積分を計算してみましょう。例：

```
sage: integral(x^2,x,0,2)
8/3
sage: integral( sin(x)/x ,x)
-1/2*I*Ei(I*x) + 1/2*I*Ei(-I*x)    # Eiは指数積分という特殊関数
```

```
sage: integral( sin(x)/x, x, 0, oo)
1/2*pi
```

ここで $\mathrm{Ei}(x)$ は指数積分 (exponential integral) と呼ばれる特殊関数で

$$\mathrm{Ei}(x) := \mathrm{p.v.} \int_{-\infty}^{x} \frac{e^t}{t} dt := \lim_{\varepsilon \to +0} \left(\int_{-\infty}^{-\varepsilon} + \int_{\varepsilon}^{x} \right) \frac{e^t}{t} dt$$

で定義されるものです。もとの関数の積分より少し複雑になったといえるかもしれません。一方，定積分 $\int_0^\infty \frac{\sin x}{x} dx$ は $\frac{\pi}{2}$ と具体的な値が求められました。

次を試してみましょう：

```
sage: integral( 1/sqrt((1-x^4)) ,x,0,1)
1/4*beta(1/4, 1/2)    # 積分値はベータ関数を用いて表される
sage: n(1/4*beta(1/4),1/2))
1.31102877714606
```

ここで beta はベータ関数

$$\beta(x, y) = \int_0^1 t^{x-1}(1-t)^{y-1} dt$$

を意味しています。上の例のようにベータ関数の値の近似値を計算することができます。

次の積分はプランクの輻射公式から真空のエネルギー密度を計算するときに現れます。

$$\int_0^\infty \frac{x^3}{e^x - 1} dx = \frac{\pi^4}{15} \tag{8}$$

これを求めてみましょう。

```
sage: f = x^3/(e^x-1)          # 関数fを定義
sage: integral(f,x,0,oo)       # f(x)を0から∞まで積分
-1/15*pi^4 + limit(-1/4*x^4 + x^3*log(-e^x + 1) + 3*x^2*dilog(e^x) - 6*x*
polylog(3, e^x) + 6*polylog(4, e^x), x, +Infinity, minus)
```

これでは，とても積分値が求まったとはいえませんが，積分のオプションをつける事によりシンプルな値が求まります。

```
sage: integrate(f,x,0,oo,'giac')    # オプション'giac'を付す。
 1/15*pi^4
```

Sage は多くの数学ソフトウェアを組み合わせて作られており，どのソフトウェアのアルゴリズムを使うかにより計算結果が変わる場合があります。積分計算のアルゴリズムのオプションは次のものがあります：

- 'maxima'：デフォルト（何も指定しないとこれを使う）
- 'sympy'：sympy を使う。Sage に含まれている。
- 'mathematica_free'：Wolfram alpha を使う。要インターネット
- 'giac'：Giac を使う。
- 'fricas'：FriCAS を使う。別途インストールしてある場合に利用可能。

29.3 数値積分

integral で不定積分も定積分も求まらないような関数はたくさんあります。これは，プログラムの能力が低いためではなく，そもそも既存の関数や定数を用いて答えを表すことが出来ないという場合が多いです。

そのような場合でも定積分の近似値を得ることができます。

積分 $\int_a^b f(x)dx$ を近似的に求める

```
1 N(integral(f(x),x,a,b))
2 numerical_integral(f(x),a,b)
```

- 1 行目では定積分を代数計算してからその近似値を求めます。
- 2 行目では，区間のサンプル点をとり特定のアルゴリズムによって数値計算を行います。

結果として (値，誤差) が返されます。

- numerical_integral ではオプションとして max_points により数値積分のサンプル点の最大個数を指定することができる。

例：

```
1 sage: integral(sin(1/x),x,0,1)
2 -1/2*Ei(I) - 1/2*Ei(-I) + sin(1)  # 厳密
3 sage: N(integral(sin(1/x),x,0,1))
4 0.504067061906928                 # 上の結果の数値(近似値)
5 sage: numerical_integral(sin(1/x),0,1)
6 (0.5040702199680792, 0.00012692441400447902) #(数値積分の結果,誤差)
```

上の最後の 2 つの数値は少し異なり誤差が出ていることがわかります。numerical_integral の 2 つめの数値は誤差を表しています。ただし，これは厳密な誤差ではなくアルゴリズムから予想される真の値との誤差です。ただし，数値積分の結果が信頼できるものであるかどうかは被積分関数の性質によります。振動が激しい関数や積分が緩やかに減衰する関数の場合などでは真の値と大幅に違った値になることもあります。

数値積分 numerical_integral は積分区間を分割して計算していますが，そのとき区間の分割が十分でないと誤差が大きくなります。サンプル点の最大個数はデフォルトでは 87 個ですが，オプションで変更可能です。サンプル点の数を多くして誤差を小さくすることができます：

```
1 sage: N(integral(sin(x^2),x,0,100))
2 0.631417921866929     # これは真の値に近い
3 sage: numerical_integral(sin(x^2),0,100) # 数値積分
4 (0.0967129340696719, 22.541102691447605)  # 誤差が大きすぎる
5 sage: numerical_integral(sin(x^2),0,100,max_points=1000)
6 (0.6314179218667048, 5.968439387227933e-07) # 誤差が減った
```

numerical_integral は GNU Scientific Library(GSL) という数値計算用の C のアルゴリズムを用いていますが，数式処理ソフト maxima を用いて数値積分を行うこともできます。こちらでは，要求する精度を指定して積分を計算することができます。

$\int_a^b f(x)dx$ の数値積分 (maxima を使用)

```
f.nintegral(x,a,b,精度)
```

実行結果は（値，誤差，近似する区間の数，エラーコード）となる。エラーコードは 0 から 6 まであり次を意味する。

- 0：エラーなし
- 1：分割する区間が多すぎる
- 2：丸め誤差が大きすぎる
- 3：被積分関数の振る舞いが最悪
- 4：収束しない
- 5：積分はおそらく発散するもしくはゆっくり収束する
- 6：入力が正しくない

例:

```
sage: f=sin(1/x)
sage: f.nintegral(x,0,1)
(0.50411061321101469, 3.4994456304171528e-05, 8379, 1)
```

積分値は以前の計算とは 5 桁めが異なります。被積分関数の振動が激しいのでエラーメッセージ 1 がでています。つぎに精度を 1/100 として計算してみます：

```
sage: f.nintegral(x,0,1,1/100)    # 精度は1/100と指定
(0.50279836327622895, 0.0046840581358000843, 567, 0)
```

最後の出力のエラーコードは 0 なので誤差 0.01 で確からしい値が得られたことがわかります。

30　Taylor 展開

30.1　Taylor 展開

f を適当な回数だけ微分可能な関数とする。f の Taylor 展開 (または Taylor 級数) とは

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-1) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n + \cdots$$

のことです。特に $a = 0$ の場合を Maclaurin 展開といいます。Taylor 展開は次のように求めます：

Taylor 級数：$f(a) + \frac{f'(a)}{1!}(x-1) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n$

関数 `f(x)` の `x=a` の周りの `x` についての `n` 次までの Taylor 級数を返す：

```
taylor(f(x),x,a,n)
```

例えば，e^x の $x = 0$ のまわりの 3 次までの Taylor 級数は

```
taylor(e^x,x,0,3)
```

実行結果の例

```
1/6*x^3 + 1/2*x^2 + x + 1
```

で求めます。2 変数関数 $f(x,y)$ の Taylor 展開は

2 変数関数 $f(x,y)$ の点 (a,b) のまわりの n 次までの Taylor 級数

```
taylor(f(x,y),(x,a),(y,b),n)
```

です。例えば，

```
var('y')
taylor(sin(x+y),(x,0),(y,0),4)
```

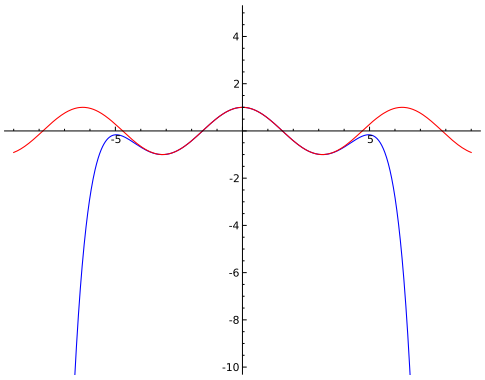
の実行結果は次のようになります：

$$-\frac{1}{6}x^3-\frac{1}{2}x^2y-\frac{1}{2}xy^2-\frac{1}{6}y^3+x+y$$

30.2　応用編：Taylor 展開で近似される様子を描画

応用編として，$\cos(x)$ とその Taylor 展開の x^{10} 次までの多項式のグラフを重ねて描いてみましょう：

```
f = taylor(cos(x),x,0,10)                  # fはcos(x)の10次までの展開
p1 = plot(f,(x,-9,9))                      # fのグラフ
p2 = plot(cos(x),(x,-9,9), color='red')    # cos(x)のグラフ
p = p1 + p2;                               # 二つのグラフを重ねたものをpとする
p.show(ymin=-10,ymax=5)                    # pを描画
```



30.3　応用編：Fourier 級数の様子を描画

区間 $[0,L]$ で定義された関数 $f(x)$ は

$$f(x)=\frac{a_0}{2}+\sum_{n=1}^{\infty}\left(a_n\cos\frac{2\pi nx}{T}+b_n\sin\frac{2\pi nx}{T}\right) \tag{9}$$

と展開されるというのが Fourier の主張です。ここで展開係数 a_n,b_n は $f(x)$ から決まる定数で

$$a_n=\frac{2}{T}\int_0^L f(x)\cos\frac{2\pi nx}{L}dx,\qquad (n=0,1,2,\cdots)$$
$$b_n=\frac{2}{T}\int_0^L f(x)\sin\frac{2\pi nx}{L}dx\qquad (n=1,2,\cdots)$$

となります。(9) の右辺は明らかに周期 L の周期関数なので，展開 (9) は $[0, L]$ で値 $f(x)$ をとる周期 L の周期関数の展開であると考えることもあります。

例：$L = 1$, $f(x) = x(1-x)$ $(0 \le x < 1)$ の場合，係数は簡単に計算できて

$$a_n = 2\int_0^1 x(1-x)\cos 2\pi n x dx = \begin{cases} \frac{1}{3} & n = 0 \\ -\frac{1}{\pi^2 n^2} & n = 1, 2, \cdots \end{cases}$$

$$b_n = 2\int_0^1 x(1-x)\sin 2\pi n x dx = 0 \qquad n = 1, 2, \cdots$$

となりますが，あえて Sage で計算してみます：

```
var('k')    #nのかわりにkを使う
assume(k, 'integer')  #kを整数とする
integral(2*x*(1-x)*cos(2*pi*k*x),x,0,1).show()  #a_k
integral(2*x*(1-x)*sin(2*pi*k*x),x,0,1).show()  #b_k
```

（Jupyter ノートブックでない場合は `.show()` を削除してください）。結果は

$$0, \qquad -\frac{1}{\pi^2 k^2}$$

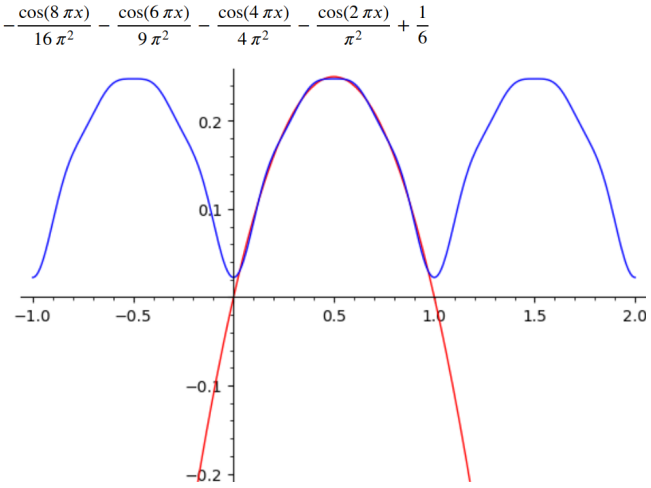
となり，$k = 0$ に対して間違えています[*16]。正しくは $a_0 = 2\int_0^1 x(1-x)\cos 0 dx = \frac{1}{3}$ です。

> コンピューターによる積分は Sage に限らず信頼しすぎないほうがよいです。計算機代数システムでは Mathematica が有名で，実際，かなり優秀な積分機能を持っているのですが，それでも積分が正しいかどうか疑った方がよいです[a]。特にパラメーターを含む積分で 1/0 が現れる場合には要注意です。
>
> [a] 古いバージョンでは，誤った積分を返す例が知られていました。最新バージョンについては把握していません。

さて，$f(x) = x(1-x)$ $(0 \le x < 1)$ とその Fourier 級数の部分和をプロットしてみましょう。級数 (9) の和を 4 個までとった関数を $g(x)$ とします。

```
var('k')    #nのかわりにkをつかう
N = 4
f=x*(1-x)
a(k) = -1/(pi^2*k^2)  #a_k for k=1,2,...
g = 1/6 + sum([a(k)*cos(2*pi*k*x) for k in range(1,N+1)])  #Fourier級数の部分和
g.show()
p1 = plot(f,x,-1,2,color='red') # f(x)=xのグラフ
p2 = plot(g,x,-1,2)             # Fourier級数の部分和のグラフ
(p1+p2).show(ymin=-0.2)
```

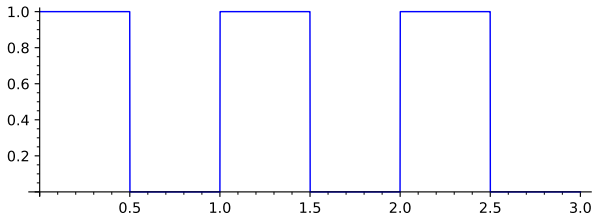
[*16] Sage 9.4 の結果です。



例：周期 1 の周期関数で条件

$$f(x)=\begin{cases}1 & 0\le x\le 1/2\\ 0 & 1/2<x<1\end{cases}$$

を満たす関数を考えます。この関数 $f(x)$ は矩形波と呼ばれ次のグラフのような形をしています。



$L=1$ の Fourier 級数 (9) を考えます。係数 a_n, b_n は，$a_0=2\int_0^1 f(x)dx=1$

$$a_n=2\int_0^{1/2}\cos 2\pi nx dx=0,\qquad n=1,2,\cdots$$
$$b_n=2\int_0^{1/2}\sin 2\pi nx dx=\begin{cases}0 & n\text{:偶数}\\ \frac{2}{\pi n} & n\text{:奇数}\end{cases}$$

ですので，(9) は

$$f(x)=\frac{1}{2}+\sum_{k=1}^{\infty}\frac{2}{\pi(2k-1)}\sin\Big(2\pi(2k-1)x\Big)\tag{10}$$

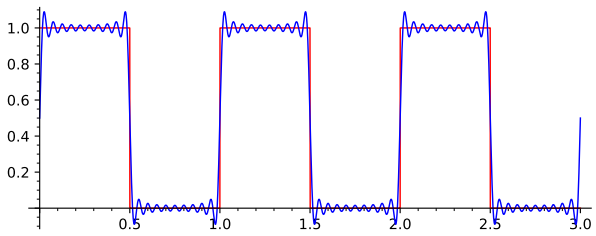
となります。この右辺を有限項にしてプロットしてみましょう。矩形波の関数 $f(x)$ を Sage で表す方法はいろいろ考えられますが，$x\ge 0$ では 1，$x<0$ では 0 となる関数 `unit_step(x)` を使って `f(x)=unit_step(sin(2*pi*x))` と定義します[*17]。

```
1 f(x) = unit_step(sin(2*pi*x))
2 N=10
3 g = 1/2 + (2/pi)*sum([sin(2*pi*(2*k-1)*x)/(2*k-1) for k in range(1,N+1)])
```

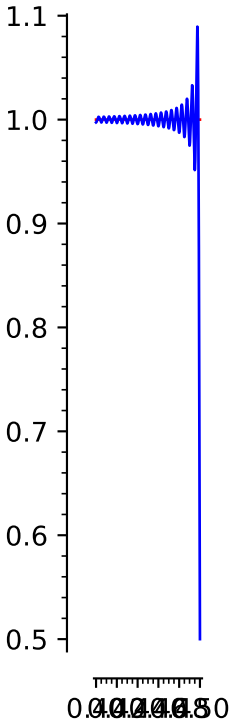
[*17] 他には `sgn(floor(x)-x+1/2)/2+1/2` も同じような関数です。ここで `floor(x)` は x の整数部分を返す関数，`sgn(x)` は x の符号 $-1, 0, 1$ を返す関数です。

```
4 p1 = plot(unit_step(sin(2*pi*x)),x,0,3,color='red')
5 p2 = plot(g,x,0,3,aspect_ratio=1)
6 (p1+p2).show()
```

上の実行結果は次のようになります。



矩形波の Fourier 級数の式 (10) は不連続点以外では正しいことが証明されています。不連続点の近くでの振る舞いが気になるので `N=100` として $[0.4, 0.5]$ の近傍だけをプロットすると次のグラフのようになっていることがわかります：



グラフから値が収束しない点 x が不連続点 $x = 1/2$ の近くに現れていることが読み取れます。このような現象を Gibbs 現象といいます。この関数の場合，そのような点における誤差は，N をいくら大きくしても

$$\frac{1}{\pi}\int_0^{\pi}\frac{\sin x}{x}dx - \frac{\pi}{2} \fallingdotseq 0.08949$$

より小さくはならないことが Gibbs によって証明されました (1899 年)。

31 練習問題

次の極限を求めよ。

$$(1)\quad \lim_{h\to 0}\frac{(x+h)^3-x^3}{h} \qquad (2)\quad \lim_{x\to\infty}\left(\frac{x^x}{x!}\right)^{1/x} \qquad (3)\quad \lim_{x\to\infty}\arctan(x)$$

次の式を部分分数展開せよ。ただし $x!$ は `factorial(x)`。

$$(4)\quad \frac{3x-37}{(x+1)(x-4)} \qquad (5)\quad \frac{9-9x}{2x^2+7x-4} \qquad (6)\quad \frac{4x^2}{(x-1)(x-2)^2} \qquad (7)\quad \frac{8x^2-12}{x(x^2+2x-6)}$$

次の数の 11 次までの連分数展開を求めよ。

$$(8)\quad \text{黄金比} = \mathsf{golden\_ratio} \qquad (9)\quad \text{円周率} = \texttt{pi}$$

次の関数を $n=1,2,3,4$ に対して計算せよ:

(10) $H_n(x)=(-1)^n e^{x^2}\dfrac{d^n}{dx^n}e^{-x^2}$ (エルミート多項式)

次の x についての関数の原始関数を求めよ。ただし $a>0$ とする。

$$(11)\quad a^x \qquad (12)\quad \frac{1}{\sqrt{a^2+x^2}}$$

次の関数の右に書かれている区間についての定積分を求めよ。

(13) $x\cos(x)$，区間 $[0,\pi]$

次の関数について，$x=0$ のまわりで 5 次までのテイラー級数をもとめよ。

(14) x^2e^x

(15) $\sin(x)/(1+x^2)$

上の問題を Sage に計算させてみましょう。計算は jupyter ノートブックで行い，ファイル名 `sage02.ipynb` としましょう。

32　微分方程式

32.1　常微分方程式とベクトル場

次の微分方程式を考えよう

$$\frac{dy}{dx} = y - x. \tag{11}$$

これの一般解は任意定数 C を用いて $y(x) = x + 1 + Ce^x$ と表される。次に微分方程式 (11) を初期条件

$$y(0) = \frac{2}{3} \tag{12}$$

の元で解いてみよう。上の一般解に $x = 0$ の値を代入し，$y(0) = 1 + C = 2/3$ から C を求めると $C = -1/3$ となることがわかる。よって，この初期値問題の解は

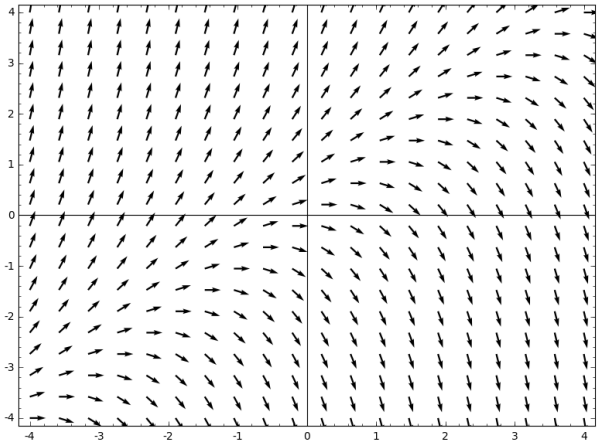
$$y(x) = x + 1 - \frac{1}{3}e^x \tag{13}$$

である。このようにして，いくつかの微分方程式については，代数的計算によって微分方程式の解を求めることができる。

コンピューターによる描画を用いることで，解の様子を視覚的に観察してみよう。

微分方程式 (11) の解曲線 $y = y(x)$ は平面上の点 (x, y) で傾き $y - x$ を持つ。そこで，点 (x, y) に $(1, y - x)$ の向きの単位ベクトルが生えているベクトル場 $\vec{V}(x, y)$ を描いてみよう。そのようなベクトル場は勾配場 (slope field) と呼ばれる。$\vec{V}(x, y) \propto (1, y - x)$（比例）なのでベクトル場 $\vec{V}(x, y) = (m, m(y - x))$ を描けばよい。ここで $m = (1 + (y - x)^2)^{-1/2}$ は $V(x, y)$ を単位ベクトルにするための規格化定数である。

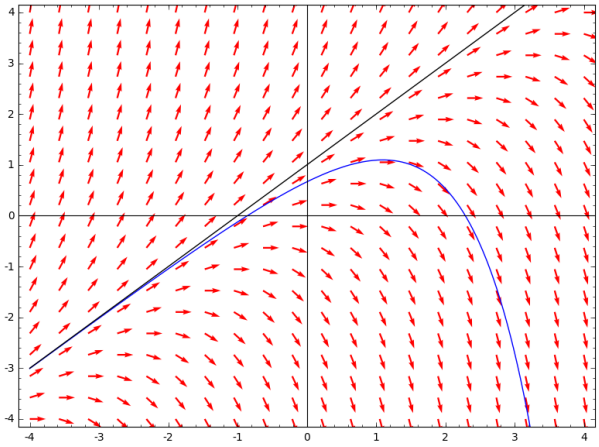
```
var('y')
m = sqrt(1+(y-x)^2)^(-1)
plot_vector_field( (m,m*(y-x)), (x,-3,3), (y,-3,3) )
```



微分方程式 (11) の解は，初期条件 $(0, y(0))$ から出発して，上の絵の矢印をその向きに歩くことで得られる。上の絵を見れば，解の振る舞いは一目瞭然で，直線 $y = x + 1$ を境に，その上側と下側で $x \to \infty$ のときの行き先が異なることがわかる。この勾配場 (赤) に $y = x + 1$ の直線 (黒) および解 (13)(青) を重ねてみよう。

```
var('y')
m = sqrt(1+(y-x)^2)^(-1)
```

```
vecf = plot_vector_field( (m,m*(y-x)), (x,-4,4), (y,-4,4) ,color="red")
sol = plot(x+1-(1/3)*e^x, (x,-4,4))        #exact solution
asym = plot(x+1, (x,-4,4), color='black') # 漸近曲線
(vecf+sol+asym).show(ymin=-4, ymax=4)
```



32.2 Euler 法

微分方程式の多くは代数的に解くことはできないが，その初期値問題の解を数値的に計算することは可能である。ここでは，微分方程式の数値計算法の一つである Euler 法について解説する。$f(x,y)$ を与えられた関数として，微分方程式

$$\frac{dy}{dx} = f(x,y) \tag{14}$$

及び初期条件 $y(x_0) = y_0$ を考える。(x_0, y_0) は与えられた数値である。上の微分方程式は無限小 dx を用いて

$$y(x+dx) = y(x) + f(x,y)dx \tag{15}$$

と書くことができる。Euler 法は無限小 dx を小さい数 Δx に置き換えることで解を近似しようとするものである。つまり，初期値 (x_0, y_0) とするとき，$x_1 := x_0 + \Delta x$ での y の値は

$$y_1 = y_0 + f(x_0, y_0)\Delta x \tag{16}$$

に非常に近いはずである。次に $x_2 := x_1 + \Delta x$ での y の値は

$$y_2 = y_1 + f(x_1, y_1)\Delta x \tag{17}$$

で近似される。同様にして $x_n := x_0 + n\Delta x$ での y の値は

$$y_n = y_{n-1} + f(x_{n-1}, y_{n-1})\Delta x \tag{18}$$

によって近似される。

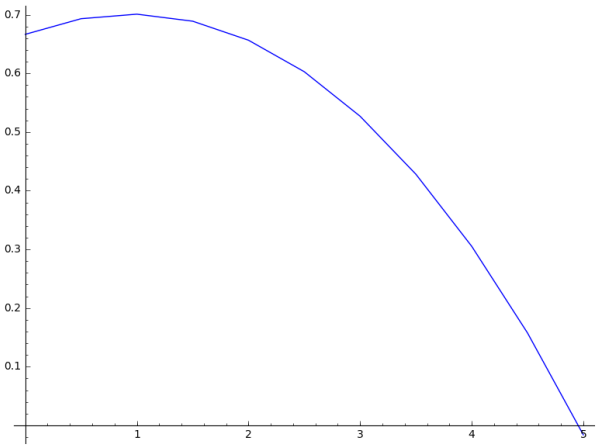
Euler 法の手順にしたがって微分方程式の初期値問題 (11), (12) の解 $y(x)$ を近似してみよう。以下では，Δx と n の値を適当に指定して，リスト $((x_0, y_0), (x_1, y_1), \cdots, (x_n, y_n))$ を作る。

```
x0, y0 = 0, 2/3    # 初期条件
Dx= 0.5            # Delta xを設定
var('y')
```

```
f(x,y) = y-x        # f(x,y)を定義
SOL = [(x0,y0)]             # 初期条件だけからなるリスト
for j in range(1,11):
    X = x0 + Dx             # 次のxの値
    Y = y0 + f(x0,y0)*Dx    # 次のyの値
    SOL.append((X,Y))       # (X,Y)をリストに加える
    x0,y0 = X, Y            # x0, y0をX,Yに置き換える
print(SOL)                  # 解曲線を近似したリストをプリント
approx = list_plot(SOL,plotjoined=True)    # SOLをプロット
approx.show()
```

実行結果の例

```
y
[(0, 2/3), (0.500000000000000, 1.00000000000000), (1.00000000000000, 1.25000000000000),
(1.50000000000000, 1.37500000000000), (2.00000000000000, 1.31250000000000),
(2.50000000000000, 0.968750000000000), (3.00000000000000, 0.203125000000000),
(3.50000000000000, -1.19531250000000), (4.00000000000000, -3.54296875000000),
(4.50000000000000, -7.31445312500000), (5.00000000000000, -13.2216796875000)]
```

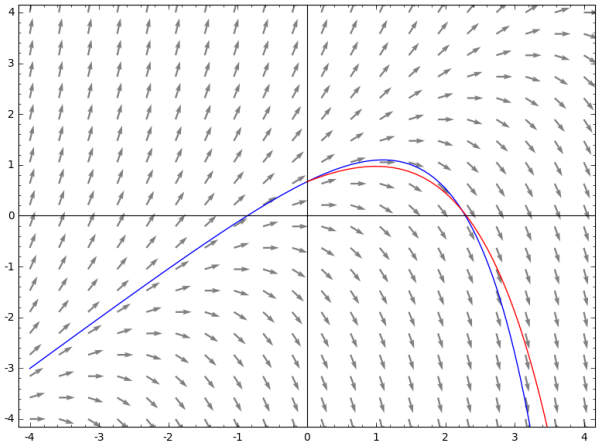


Euler 法による近似解と真の解とを比較してみよう。

```
var('y')
m = sqrt(1+(y-x)^2)^(-1)
vecf = plot_vector_field( (m,m*(y-x)), (x,-4,4), (y,-4,4) ,color="gray")
sol = plot(x+1-(1/3)*e^x, (x,-4,4))    # 真の解のグラフ

x0, y0 = 0, 2/3    # 初期条件をセット
Dx= 0.05         # Delta xを設定
f(x,y) = y-x        # f(x,y)を定義
SOL = [(x0,y0)]             # 初期条件だけからなるリスト
for j in range(1,81):
    X = x0+Dx               # 次のxの値
    Y = y0 + f(x0,y0)*Dx    # 次のyの値
    SOL.append((X,Y))       # (X,Y)をリストに加える
    x0,y0 = X, Y            # x0, y0をX,Yに置き換える
approx = list_plot(SOL,plotjoined=True,color='red')    # SOLをプロット

(vecf+sol+approx).show(ymin=-4, ymax=4) # 赤線がEuler法によって計算したもの
```



Euler 法によって計算した値は，厳密な値のグラフと比べると誤差があるのがわかる。この誤差は Δx を小さく取ることにより，小さくすることができる。例えば，上のプログラムで `Dx=0.001` としてみよう。（このとき 10 行目の `range(1,81)` を変えて，計算のステップ数も増やす必要がある。）

32.3 微分方程式の厳密解

この節では SageMath の機能である `desolve` 関数を使って微分方程式を代数的に解きます。`desolve` の文法とオプションは次の通りです：

微分方程式を解く (desolve)

```
1 desolve(de, dvar, options)
```

ここで，

- `de`: 微分方程式 (differential equation)
- `dvar`: 従属変数 (dependent variable:求める関数)

です。オプション `options` は次が用意されています：

- `ics`: 初期条件 (initial conditions) もしくは境界条件を指定
- `ivar`: 独立変数 (independent variable)
- `show_method`: 解法に使用した手法 (微分方程式の型) を表示する
- `contrib_ode`: Clairaut, Lagrange, Riccati 型の方程式も含めて解を探す場合に `True` にする。オプションは省略可能です。

さて簡単な微分方程式

$$y'(x) = y(x) \tag{19}$$

を解いてみましょう。これ解くには次のようにします：

```
1 y = function('y')(x)           # yをxの関数とする
2 desolve( diff(y,x) == y, y)    # 微分方程式 y'=y をyについて解く
```

実行結果の例

```
_C*e^x
```

上のプログラムの 2 行目で，y は x の関数であるという宣言をしています。3 行目の diff(y,x) は微分 $y'(x)$ を意味していて，diff(y,x)==y が解きたい微分方程式ということになります。実行結果の_C は任意定数です。

desolve では微分方程式を指定するときに==0 を省略することができます。例えば，desolve(diff(y,x)-y, y) と desolve(diff(y,x)-y == 0, y) は同じ意味です。

```
y = function('y')(x)
desolve( diff(y,x) - y , y)   # 微分方程式 y'-y =0 を yについて解く
```

実行結果の例

```
_C*e^x
```

32.3.1 微分方程式の解法 1

次の微分方程式を解く事を考えましょう。

$$y'(x) + 2y(x)\sin(x) = 0 \tag{20}$$

微分方程式を勉強した事がある人は，変数分離型の解法によって直ちにこれを解く事ができると思います。忘れた人もいるかもしれないので念のために変数分離型の解法を復習しておきます。まず上の微分方程式を

$$\frac{y'}{y} = -2\sin(x) \tag{21}$$

と変形して[*18]，両辺を x で積分する事により

$$\log y = \int \frac{y'}{y} dx = 2\cos(x) + c \tag{22}$$

となります。これを y について解くことにより答えは

$$y(x) = \exp(2\cos(x) + c) \tag{23}$$

となります。これも一般解といえますが，これでは y が恒等的に 0 となる自明な解を含みません。そこで，$C = e^c$ を新たに任意定数とすると，(20) の解は

$$y(x) = Ce^{2\cos(x)} \tag{24}$$

となります。さて，上の微分方程式を Sage で解いてみましょう：

```
y = function('y')(x)
DE = diff(y,x) + 2*y*sin(x) == 0 # 微分方程式を DEと名付ける
desolve(DE, y)                   # DEをyについて解く
```

実行結果の例

```
_C*e^(2*cos(x))
```

32.3.2 微分方程式の解法 2

次の微分方程式を考えます：

$$y'(x) = \frac{x - y(x)}{x + y(x)}. \tag{25}$$

これも変数分離型で解く事ができますが，Sage に解かせてみましょう：

[*18] $y(x) = 0$ のときはどうするんだと指摘されそうですが，とりあえず $y(x) \neq 0$ と仮定しましょう。

```
y = function('y')(x)
desolve( diff(y,x) == (x-y)/(x+y), y)
```

実行結果の例

```
-1/2*x^2 + x*y(x) + 1/2*y(x)^2 == _C
```

微分方程式 (25) の解 $y(x)$ は，微分を含まない方程式

$$-\frac{1}{2}x^2 + xy(x) + \frac{1}{2}y(x)^2 = C \tag{26}$$

を満たす関数として陰 (implicitly) に解かれました。上式は $y(x)$ についての 2 次関数なので解の公式を使って直ちに解く事ができますが，Sage の solve にそれをやらせてみましょう：

```
y = function('y')(x)
SOL = desolve( diff(y,x) == (x-y)/(x+y), y)   # 微分方程式の答えをSOLと名づける
solve(SOL,y)                                  # 方程式SOLをyについて解く
```

実行結果の例

```
[y(x) == -x - sqrt(2*x^2 + 2*_C), y(x) == -x + sqrt(2*x^2 + 2*_C)]
```

したがって，微分方程式の答えは，$y(x) = -x \pm \sqrt{2x^2 + 2C}$ となることが分かります。

定数 C と符号 $\pm$ に応じて，微分方程式の解は無数にありますが，これをグラフに描いてみましょう：

```
p = Graphics()     # 空のグラフィックスオブジェクトを作る
for C in range(4):           # 定数Cを0から3まで変えながら解のグラフをpに加える
  p += plot( -x - sqrt(2*x^2 + 2*C), (x,-3,3),
        color=hue(C/4,1 ), legend_label='C='+str(C)+',$+$') #Cごとに色を変える
  p += plot( -x + sqrt(2*x^2 + 2*C), (x,-3,3),
        color=hue(C/4,0.5), legend_label='C='+str(C)+',$-$')#Cごとに色を変える

p.show()      # pを描画
```

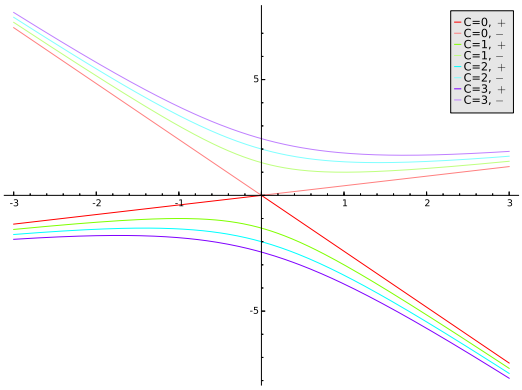


図 33　出力結果：微分方程式 (25) の解のグラフ

32.3.3　微分方程式の解法 3

次の微分方程式を考えます：

$$y' = \sin(x + y) + \sin(x - y) \tag{27}$$

これも加法定理を使えば変数分離型にできて簡単に解く事ができますが，このままでは Sage は解く事ができません，

```
y = function('y')(x)
desolve( diff(y,x) == sin(x+y)+sin(x-y), y)
```

実行結果の例

```
NotImplementedError: Maxima was unable to solve this ODE. Consider to
set option contrib_ode to True.
```

あらかじめ右辺を加法定理で因数分解しておけば解く事ができます：

```
y = function('y')(x)
DE = diff(y,x) == sin(x)*cos(y)+sin(y)*cos(x)+sin(x)*cos(y)-sin(y)*cos(x)
desolve(DE, y)
```

実行結果の例

```
-1/4*log(sin(y(x)) - 1) + 1/4*log(sin(y(x)) + 1) == c - cos(x)
```

32.4　二階の微分方程式

$y(x)$ の二階までの微分を含む方程式を二階の微分方程式といいます。x の関数 y の二階微分は diff(y,x,2) です。通常，二階の微分方程式の解は 2 つの任意定数を持ちます。それでは，二階の微分方程式

$$y''(x) - 8y'(x) + 15y(x) = 0 \tag{28}$$

を解いてみましょう。

```
y = function('y')(x)
desolve( diff(y,x,2)-8*diff(y,x)+15*y == 0, y)
```

実行結果の例

```
_K1*e^(5*x) + _K2*e^(3*x)
```

ここで_K1，_K2 は二つの任意定数です。

32.5　オプション 1 （Riccati, Clairaut, Lagrange を含めた解法）

desolve は，さまざまなタイプの微分方程式を解くことができますが，Riccati 型微分方程式

$$\frac{dy}{dx} + a(x)y^2 + b(x)y + c(x) = 0$$

や Clairaut 型の微分方程式

$$y = x\frac{dy}{dx} + f\left(\frac{dy}{dx}\right)$$

や力学の講義で学ぶ Lagrange 方程式の可能性も含めて解かせるにはオプションで contrib_ode=True と指定する必要があります。このとき，どの手法を用いたかを知るためには，オプションとして show_method=True を指定します。

では微分方程式 $y'^2 + xy' - y = 0$ を解いてみましょう：

```
y = function('y')(x)
DE = diff(y,x)^2 + x*diff(y,x)-y == 0      # 微分方程式DEを定義
desolve(DE, y, contrib_ode=True, show_method=True)
```

実行結果の例

```
[[y(x) == _C^2 + _C*x, y(x) == -1/4*x^2], 'clairault']
```

上の微分方程式を解くときに Clairaut 型の微分方程式の解法が使われたことがわかります。

32.6　オプション２：方程式が定数を含む場合

$y'(x) = ay(x)$ のように微分方程式が定数 a を含む場合は，その変数を `var('a')` で指定するのと同時に，オプションとして独立変数を `ivar=x` と指定します。

```
var('a')      # aを変数とする
y = function('y')(x)
desolve(diff(y,x) + a*y == 0, y, ivar=x)
```

実行結果の例

```
_C*e^(-a*x)
```

微分方程式 $y'' = ay$ は a が正か負か 0 かによって解が劇的に変化します。a の符号がわからないと微分方程式が解けない場合があります：

```
var('a')
y = function('y')(x)
desolve(diff(y,x,2) + a*y == 0, y, ivar=x)
```

実行結果の例

```
Traceback (click to the left of this block for traceback)
...
Is a positive, negative or zero?
```

このようなメッセージが出た場合，a に対する条件を指定すると解けるかもしれません。例えば $a > 0$ と仮定すると次のようになります：

```
var('a')
y = function('y')(x)
assume(a>0)           # a>0と仮定
desolve(diff(y,x,2) + a*y == 0, y, ivar=x)
```

実行結果の例

```
_K2*cos(sqrt(a)*x) + _K1*sin(sqrt(a)*x)
```

32.7　オプション３：初期条件と境界条件 (ics)

$y' = y$ の一般解は $y = Ce^x$ です。初期条件が $y(0) = 1$ なら $C = 1$ です。一階の微分方程式を初期条件

$$y(x_0) = y_0 \tag{29}$$

の下で解くときには，オプション (`ics`) に $[x_0, y(x_0)]$ を指示します。例えば微分方程式 $y' = y$ を初期条件 $y(0) = 1$ で解くなら次のようにします。

```
y = function('y')(x)
desolve( diff(y,x) + y == 0, y, ics=[0,1] )   # 初期条件icsを指定
```

実行結果の例

```
e^x
```

二階の微分方程式の初期条件は `ics`$= [x_0, y(x_0), y'(x_0)]$ のように指定します。

例えば微分方程式 $y'' + 2y' + y = 0$ で，初期条件を $y(0) = 3, y'(0) = 1$ とするなら

```
y = function('y')(x)
desolve( diff(y,x,2) + 2*diff(y,x) + y , y, ics=[0,3,1] ) # 初期条件を指定
```

実行結果の例

```
(4*x + 3)*e^(-x)
```

とします。また境界条件[*19]を設定するには ics=$[x_0, y(x_0), x_1, y(x_1)]$ をオプションとして書きます。上と同じ微分方程式を境界条件 $y(0) = 3, y(\pi/2) = 2$ として解くと

```
y = function('y')(x)
desolve( diff(y,x,2)+2*diff(y,x)+y , y, ics=[0,3,pi/2,2] ) # 境界条件を指定
```

実行結果の例

```
(2*(2*e^(1/2*pi) - 3)*x/pi + 3)*e^(-x)
```

となります。

32.8　その他の方法

微分方程式 $xy' - x\sqrt{y^2 + x^2} - y = 0$ はそのままでは解けない：

```
y = function('y')(x)
desolve( x * diff(y,x) - x * sqrt(y^2+x^2) - y == 0, y, contrib_ode=True )
```

実行結果の例

```
Traceback (click to the left of this block for traceback)
...
TypeError: ECL says: Maxima asks: Is  y  zero or nonzero?
```

しかし，x と y の範囲に条件を付ければ解ける：

```
y = function('y')(x)
assume(x>0);  assume(y>0);          #x>0, y>0 と仮定する
desolve( x * diff(y,x) - x * sqrt(y^2+x^2) - y == 0, y, contrib_ode=True )
```

実行結果の例

```
x - arcsinh(y(x)/x) == c
```

上の結果は Sage4.6 の結果ですが，Sage6.7 では次のようになりました。

実行結果の例

```
[1/2*(2*x^2*sqrt(x^(-2)) - 2*x*sqrt(x^(-2))*arcsinh(y(x)/sqrt(x^2)) -
2*x*sqrt(x^(-2))*arcsinh(y(x)^2/(sqrt(y(x)^2)*x)) +
log(4*(2*x^2*sqrt((x^2*y(x)^2 + y(x)^4)/x^2)*sqrt(x^(-2)) + x^2 +
2*y(x)^2)/x^2))/(x*sqrt(x^(-2))) == _C]
```

この左辺に simlify_full() を施せばもう少し単純な形になりますが，旧バージョンの結果ほどきれいにはなりませんでした。おそらく Sage のバージョンによって使用しているアルゴリズムが変更されたために，見た目が異なる結果が得られたのだと思われます[*20]。

33　インタラクティブな操作：@interact

Sage ノートブックの独自の機能の一つに@interact があります。これによって，式やグラフィックスなどをマウスを使ってインタラクティブに操作する事ができるようになります。これは Mathematica での Manipulate 機能に相当するものです。

簡単な例を使って@interact の使い方を紹介します。関数 f(x) を，引数 x に対して x^2 をプリントする関数であると定義します：

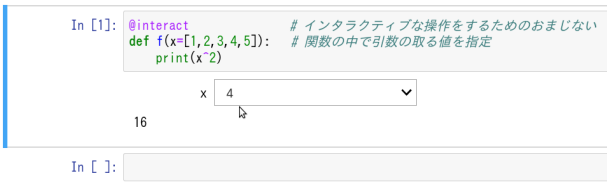
```
def f(x):          # 関数 f を定義
```

[*19] 境界 x_0, x_1 での値 $y(x_0)$ と $y(x_1)$ を指定することを境界条件を決めるという。
[*20] Sage9.1 だとまた違う結果になるようです。

```
2     print(x^2)     # return ではなく print であることに注意
```

次に，この関数に対してインタラクティブな操作を行ってみましょう。Sage ノートブックで次を実行してみましょう：

```
1 @interact          # インタラクティブな操作をするためのおまじない
2 def f(x=[1,2,3,4,5]):    # 関数の中で引数の取る値を指定
3     print(x^2)
```



x の横の枠の数字を選ぶと，その下の実行結果が変わります。

上のものは最も簡単なインタラクティブな操作の例ですが，一般には次の手順で行います：

@interact の使い方

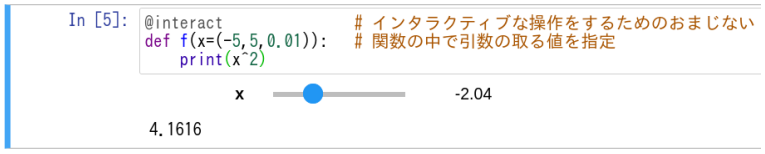
1. 実行したい処理を一つの関数として定義する
2. 関数の定義の直前に`@interact` と書く
3. 変化させたいパラメーターを関数の引数とする
4. パラメーターが変化する範囲を『引数＝』で指定する

パラメーターがとる範囲は次のように指定することができる：

- `[1,2,3,4,5]`：リスト
- `(1..5)`：1 から 5 までの自然数：(操作はスライダー)
- `(2, 10, 2)` ：2,4,6,8,10 をとるスライダー
- `(5, 10, 0.2)`：5 から 10 までの 0.2 刻みの実数（操作はスライダー）
- `slider(1, 8, default=5)`：1 から 8 までの実数をとるスライダーで，初期位置は 5　⇒
- `input_box(type=str)`：文字を入力する

先ほどの例で，次のようにするとスライダーの取る範囲が-5 から 5 までの 0.1 刻みの実数になります。

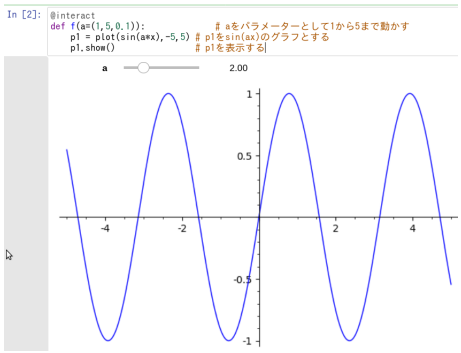
```
1 @interact
2 def f(x=(-5,5,0.01)):
3     print x^2
```



`@interact` はパラメーターを含むグラフを描いて様子を見たいときに非常に便利です：

```
1 @interact
2 def f(a=(1,5,0.1)):          # aをパラメーターとして1から5まで動かす
```

```
    p1 = plot(sin(a*x),-5,5) # p1をsin(ax)のグラフとする
    p1.show()                # p1を表示する
```



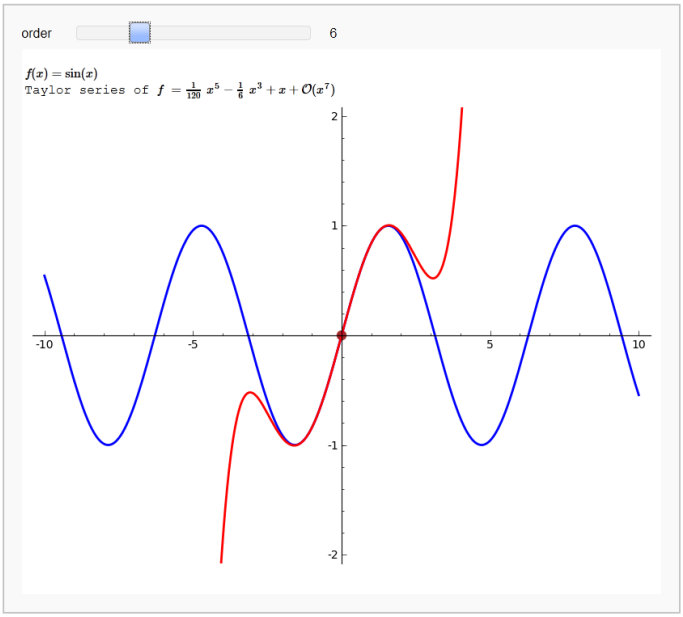
$\sin(x)$ とテイラー級数を同時に描いてみましょう：

```
x0 = 0
f  = sin(x)
p  = plot(f,-10,10)  # pはsin(x)のグラフ

@interact      # 以下で定義する関数をインタラクティブに操作
def show_taylor(nn=(1..20)):
    ft = f.taylor(x,x0,nn)                #ftはfのnn次のテイラー級数
    pt = plot(ft,-10, 10)                 #ftのグラフをptとする
    show(p + pt, ymin = -2, ymax = 2)     # グラフを描画
```

次は，上のグラフの色を変えたりオプションを様々に追加したものです。

```
x0  = 0
f   = sin(x)
p   = plot(f,-10,10, thickness=2)  # pはsin(x)のグラフ
dot = point((x0,f(x=x0)), pointsize=80,rgbcolor='brown')   # 原点に点を打つ
@interact      # 以下で定義する関数をインタラクティブに操作
def show_taylor(nn=(1..20)):
   ft = f.taylor(x,x0,nn)                                  #ftはfのテイラー級数
   pt = plot(ft,-10, 10, color='red', thickness=2) #ftのグラフをpt
   pretty_print( html('$f(x) = %s$'%latex(f)) )
   pretty_print(html('Taylor series of $f$ $= %s+O(x^{%s})$'%(latex(ft),nn+1)))
   show(dot + p + pt, ymin = -2, ymax = 2)     # グラフを描画
```



上のプログラムの 10 行目の `%s` は直後の `latex(f)` に置き換わります。同じく，11 行目の一つ目の `%s` は `latex(ft)`，二つ目の `%s` は `order+1` に置き換わります。この記法についての解説は省略します。

`@interact` については力学系やフラクタルなどおもしろい使い方が次の Web ページで紹介されています：

http://wiki.sagemath.org/interact

34　動画作成 (animate)

グラフィックスのオブジェクトのリストをつなぎ合わせて次のように簡単に動画（gif ファイル）を作ることができます：

```
1 def pic_sin(c):
2     return plot(sin(x+c),(x,0,2*pi))
3
4 lis1 = [ pic_sin(c) for c in srange(0,2*pi,0.1)]
5 b = animate(lis1, figsize=[3,2])
6 b.show()
7 b.show(delay=2)
8 b.show(delay=3)
```

描画には計算時間が多少かかります[*21]。オプション `delay` でコマ送りの早さを指定します。ブラウザ上で保存したい gif 動画上で右クリックして，「名前を付けて画像を保存」を選ぶことで，gif ファイルを保存することができます。

[*21] CoCalc では動作しないようです。Sage Cell Server で実行してみましょう。

34.1 高木関数の描画

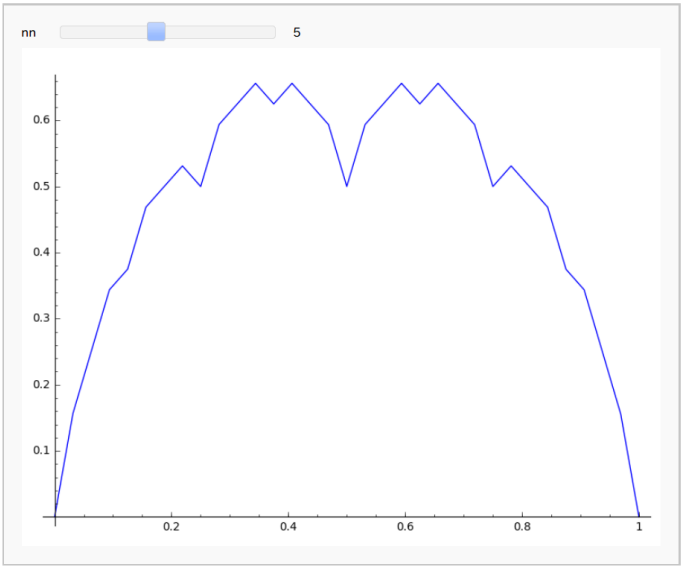
高木関数とは

$$T(x) = \sum_{n=0}^{\infty} \frac{s(2^n x)}{2^n} \tag{30}$$

で定義される関数である。ここに $s(x) = \min_{n\in\mathbb{Z}} |x - n|$。無限和を有限の和（最初の n 項の和）で近似することで，高木関数の様子を描画してみよう。

```
s = lambda x: abs(x-round(x))    #s(x)の定義（lambda記法）

@interact
def plotTakagi(nn=(1..10)):
    f = lambda x: sum( [s(2^k*x)/2^k for k in range(nn)] ) #T(x)の部分和
    plot(f,x,0,1).show()
```



高木関数のグラフをアニメーションにしてみよう[*22]。

```
def plotTakagi(nn):
    s = lambda x: abs(x-round(x))
    f = lambda x: sum( [s(2^k*x)/2^k for k in range(nn)] )
    return plot(f,x,0,1, legend_label='n='+str(nn))

figs = [plotTakagi(nn) for nn in range(1,17)]
animate(figs).show(delay=70)
```

高木関数の gif 動画 (`takagi.gif`) を生成する Sage プログラムは次のようになります[*23]。

```
def plotTakagi(nn):
    s = lambda x: abs(x-round(x))
```

[*22] CoCalc では実行できません。Sage Cell Server で実行してみましょう。
[*23] CoCalc では動作しません。Sage Cell Server で実行してみましょう。

```
    f = lambda x: sum( [s(2^k*x)/2^k for k in range(nn)] )
    return plot(f,x,0,1, legend_label='n='+str(nn))

figs = [plotTakagi(nn) for nn in range(1,17)]
pp = animate(figs)
pp.gif(savefile='~/takagi.gif', delay=70)  #ホームディレクトリにgifが生成される
```

34.2 練習問題

σ, ρ, β を定数とする。3 次元の位置座標 $x(t), y(t), z(t)$ に関する微分方程式

$$\dot{x}(t) = \sigma(y - x) \tag{31}$$
$$\dot{y}(t) = x(\rho - z)y \tag{32}$$
$$\dot{z}(t) = xy - \beta z \tag{33}$$

をローレンツ方程式 (Lorenz system) という。ここで $\dot{f}(t) = df/dt$ である (Newton の記法)。この解は，初期条件の微小な変化が後の解の挙動に大きな変化をもたらす「カオス的な振る舞い」をすることが知られている。

上の方程式の解を Euler 法で作り 3 次元のグラフとして表示させたい。定数の値は

$$\rho = 28, \qquad \sigma = 10, \qquad \beta = 8/3,$$

とし，初期条件は

$$x(0) = x_0 := 0, \qquad y(0) = y_0 := 1, \qquad z(0) = z_0 := 2$$

とすることにしよう。微小な時間間隔を $\Delta t = 0.01$ とするとき初期位置 $(x_0, y_0, z_0) = (0, 1, 2)$ から時間 Δt 経ったときの位置 (x_1, y_1, z_1) は次のように近似される。

$$\begin{aligned} x_1 &:= x(\Delta t) = x_0 + \sigma(y_0 - x_0)\Delta t = 0 + 10(1 - 0) * 0.01 = 0.1, \\ y_1 &:= y(\Delta t) = y_0 + x_0(\rho - z_0)\Delta t = 1, \\ z_1 &:= z(\Delta t) = z_0 + (x_0 y_0 - \beta z_0)\Delta t = 1.9466\cdots \end{aligned}$$

そして，ある時刻の位置が (x_n, y_n, z_n) であるとき，その時間 Δt 後の位置 $(x_{n+1}, y_{n+1}, z_{n+1})$ は

$$\begin{aligned} x_{n+1} &= x_n + \sigma(y_n - x_n)\Delta t, \\ y_{n+1} &= y_n + x_n(\rho - z_n)\Delta t, \\ z_{n+1} &= z_n + (x_n y_n - \beta z_n)\Delta t \end{aligned}$$

によって次々に計算することができる。このようにして解の座標の集まり $X = (x_0, x_1, \cdots, x_N), Y = (y_0, y_1, \cdots, y_N), Z = (z_0, z_1, \cdots, z_N)$ を作り，そこから，3 次元の座標点の集まり

$$SOL = [(x_0, y_0, z_0), (x_1, y_1, z_1), \cdots, (x_N, y_N, z_N)]$$

を作りこれを `list_plot` により表示させる。

次のプログラムを参考にして，プログラムを作成しよう。

```
rho=28
sigma=10
```

```
3  beta=8/3
4
5  def Lorenz(x,y,z):
6      dot_x = sigma*(y-x)
7      dot_y = x*(rho-z) - y
8      dot_z = x*y-beta*z
9      return dot_x, dot_y, dot_z
10
11 Dt = 0.01    # 微小時間間隔
12 N = 10000    # 計算するステップ数
13
14 X=[0]      #
15 Y=[1]      # 初期条件
16 Z=[2]      #
17 for j in range(N):
18     dot_x, dot_y, dot_z = Lorenz(X[j],Y[j],Z[j])
19     x = X[j] + ********
20     y = ***************
21     z = ***************
22     X.append(x)
23     Y.append(y)
24     Z.append(z)
25
26 SOL = list(zip(X,Y,Z))
27 list_plot(SOL, plotjoined=True)
```

26 行目で zip 関数を用いましたが，これは複数のリストの各成分の組を作り出す関数です。例えば

```
1 a = [1,2,3,4]
2 b = [5,6,7,8]
3 c = [9,10,11,12]
4 print( list(zip(a,b,c)) )
```

実行結果の例

```
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

35　行列計算

行列の計算は，数値計算の分野の中で重要な位置を占めています。線形微分方程式の解を計算する問題は，その多くの部分が行列計算に帰着され，Google の検索システムでは Web ページのランク付けのために，巨大な行列の固有ベクトルが計算されています。

35.1　ベクトルと行列の生成

まずは，Sage を使って手軽に行列計算を行う方法を紹介します。

ベクトル行列の定義

Sage ではベクトルと行列はそれぞれ次のように表します：

$$(1)\quad \begin{pmatrix}1\\2\end{pmatrix} = \texttt{vector([1,2])}$$

$$(2)\quad \begin{pmatrix}1&2&3\\4&5&6\end{pmatrix} = \texttt{matrix([[1,2,3],[4,5,6]])}$$

$$(3)\quad \begin{pmatrix}1&2&3\\4&5&6\end{pmatrix} = \texttt{matrix(2,3,[1,2,3,4,5,6])}$$

上では，2 種類の行列の作り方を紹介しました。一つは上の (2) のように 2 重のリストによって表現する方法で，二つ目は (3) のようにリストを型で区切って行列を生成する方法です。(3) では 2,3 と指定することで 2 行 3 列の行列を生成しています。

具体的にベクトルや行列を作ってみましょう：

```
a = vector([1,2])      # aをベクトル(1,2)とする
print(a)
```

実行結果の例

```
(1, 2)
```

```
b = matrix([[1,2],[3,4]])  # 行列bを定義
print(b)
```

実行結果の例

```
[1 2]
[3 4]
```

```
c = matrix(2,3,[1,2,3,4,5,6])    # 行列cを定義
print(c)
```

実行結果の例

```
[1 2 3]
[4 5 6]
```

上のように，行列の成分をリストの形に具体的に書いて全て指定することによって，行列を作ることもできますが，行列の成分が何らかのパターンを持つ場合には，次のように関数を使って行列を作ります。

例えば $f(i,j) = i + j$ を (i,j) 成分に持つような 5×5 行列を作ってみます。

```
def elem(i,j):       # 関数elemの定義
    return i+j^2     # i+j^2を返す

aa = matrix(5,5, elem)  # i,j成分が elem(i-1,j-1) となる行列
```

```
print(aa)
print(aa[3,4])            #  aaの4行5列目の成分
```

実行結果の例

```
[ 0  1  4  9 16]
[ 1  2  5 10 17]
[ 2  3  6 11 18]
[ 3  4  7 12 19]
[ 4  5  8 13 20]
19
```

Sage では行列の成分は 0 から数えるので，通常の行列の 1,1 成分は Sage では 0,0 成分となります。

35.2　ベクトルと行列の積

ベクトル同士の内積，行列とベクトルの積，行列同士の積は*で計算します。

ベクトルや行列の積

- ベクトル u と v の内積： u*v
- 行列 A とベクトル v の積： A*v
- 行列 A と B の積: A*B

たとえばベクトル $(3,-3,5)$ と $(1,1,-2)$ の内積は $3\times 1+(-3)\times 1+5\times(-2)=-10$ ですが，これを Sage で行うには

```
vector([3,-3,5]) * vector([1,1,-2])
```

実行結果の例

```
-10
```

のようにします。ただし，内積は複素内積ではないので注意しましょう：

```
vector([I,1]) * vector([I,1])      # I は虚数単位を表す。
```

実行結果の例

```
0           # I*I + 1*1 = -1+1=0    複素共役はとらない。
```

行列の積は次のように計算します：

```
matrix([[1,2],[-4,2]]) * matrix([[2,-1],[-3,2]])
```

実行結果の例

```
[ -4   3]
[-14   8]
```

35.3　行列に対する基本的な操作

Sage では行列に対して様々な操作を行うことができますが，代表的なものは次のものです：

名前	Sage での記号	数学的意味
行列式	det(A)	$\det(A)$
逆行列	A^-1	A^{-1}
転置行列	transpose(A)	${}^tA$
トレース	A.trace()	$\mathrm{tr}(A)$
$Ax=v$ を解く	A.solve_right(v)	$x=A^{-1}v$
階数	A.rank()	$\mathrm{rank}(A)=\dim \mathrm{Im}A$
核の次元	A.nullity()	$\dim\ker(A)$

行列式，転置はそれぞれ A.det() と A.transpose() で計算することもできます。

実際に確かめてみます。

```
var('a b c d')      #a, b, c, dを不定元とする
A= matrix(2,2,[a,b,c,d]) # 行列Aを定義する
print( A.transpose() )       # 転置行列
print('')                    # 改行
print(det(A))                # 行列式
print('')                    # 改行
print(A.trace())             # トレース
print('')                    # 改行
print(A^-1)                  # 逆行列
```

実行結果の例

```
[a c]
[b d]                                                    # 転置

-b*c + a*d                                               # 行列式

a + d                                                    # トレース

[1/a - b*c/(a^2*(b*c/a - d))          b/(a*(b*c/a - d))]    # 逆行列
[          c/(a*(b*c/a - d))             -1/(b*c/a - d)]
```

一般に行列 n 次正方行列 A の像 $\mathrm{Im}A$ と核 $\ker A$ の次元の間には

$$n = \dim \ker A + \mathrm{rank}A$$

という関係成り立つ事が知られています。実際の行列で試してみましょう。

```
mat = matrix(6,6, lambda i,j : i+j) # 行列を定義
print mat                           # 定義した行列をプリント
print mat.rank()                    # rank
print mat.nullity()                 # dim(ker A)
```

実行結果の例

```
[ 0  1  2  3  4  5]
[ 1  2  3  4  5  6]
[ 2  3  4  5  6  7]
[ 3  4  5  6  7  8]
[ 4  5  6  7  8  9]
[ 5  6  7  8  9 10]
2     #階数
4     #ker(mat) の次元
```

上の行列の定義では，無名関数を使うためにラムダ式を使いましたが，先ほどの説明した関数 elem を使って行列を定義するのと同じ事です。

35.4　特別な行列の作り方

単位行列

$$E_2 := \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \qquad E_3 := \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

はそれぞれ次のように作ることが出来ます：

```
E2 = matrix.identity(2)
E3 = matrix.identity(3)
print(E2, "\n")
print(E3)
```

実行結果の例

```
[1 0]
[0 1]

[1 0 0]
[0 1 0]
[0 0 1]
```

次のように対角成分以外が 0 である行列を対角行列といいます：

$$\begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 4 \end{pmatrix} =: \mathrm{diag}[2,1,4]$$

このような行列を Sage で作るには次のようにします。

```
A = diagonal_matrix([2, 1, 4]); A
```

実行結果の例

```
[2 0 0]
[0 1 0]
[0 0 4]
```

35.5 行列と係数環・係数体

コンピューターによる計算は，大きく分けて二つに分けられます。一つは厳密な計算で，もう一つは数値計算（近似計算）です。これら二つは目的が異なるため，同じ数学的概念 —例えば行列— を考える場合でも，計算上は異なる取り扱いをする必要があります。一般的に，厳密な計算には時間がかかるため，近似値だけが必要なのであれば，厳密さを無視して近似計算をするようにプログラムしてやる必要があります。

以下では，Sage の行列計算において，厳密な計算か数値計算かを指定するための方法を説明します。行列の積や和では，それを意識することはあまりありません。しかし，多項式を解かなければならないような固有値や固有ベクトル，ジョルダン標準形などを求める場合には，答えは厳密には求まらないけど，数値計算ならできるといった場合がでてきます。

Sage による行列計算では，その行列の成分がどこの係数環（または体）にあるかを指定することで厳密な計算か数値計算かが決まります。そして，それによって計算できること，できないことも異なってきます。

ここで設定できる環の代表的なものには次のようなものがあります：

名前	Sage の記号	Sage の英語名	意味	精度
整数	`ZZ`	Integer Ring	整数のつくる環 $\mathbb{Z}$	厳密
有理数	`QQ`	Rational Field	有理数体 $\mathbb{Q}$	厳密
代数的数	`QQbar`	Algebraic Field	$\mathbb{Q}$ の代数閉包	厳密
形式的な環	`SR`	Symbolic Ring	形式的な環（ほぼ体）	厳密
$\mathbb{Z}/n\mathbb{Z}$	`Integers(n)`	$\mathbb{Z}/n\mathbb{Z}$	n を法とした整数	厳密
位数 n の有限体	`GF(n)`	Finite Field of size n	位数 n の有限体	厳密
実数 (53bit)	`RR`	Real Field with 53 bits of precision	53 ビットの実数	近似
実数 (53bit)	`RDF`	Real Double Field	倍精度浮動小数点実数	近似
複素数 (53bit)	`CC`	Complex Field with 53 bits of precision	53 ビットの複素数	近似
実数 (400bit)	`RealField(400)`		400 ビットの実数	近似

行列が定義されている環を Sage では base ring といいます。例えば，整数の範囲で行列計算を行うのであれば base ring は整数にします。実数値で近似計算するのであれば，base ring は `RR` または `RDF` とします。より精度が必要であれば，`RealField(400)` などと指定して高精度で計算します。行列の係数がすべて実数値であっても，その固有値には複素数が現れることがあるので，そのような計算をする場合には，base ring とし

て CC などを指定します。QQbar は $\sqrt{2}$ や $\sqrt{3}$ などを含む体の事です[*24]。

行列の持っているメソッド base_ring() を使って，行列の base ring として何が指定されているのか確かめることができます：

```
mat = matrix(2,2,[1,2,3,4])        # 行列を定義
mat.base_ring()                    # matのbase ringを表示
```

実行結果の例

```
Integer Ring
```

上の行列の base ring は整数環 $\mathbb{Z}$ になっています。成分に一つでも有理数があると base ring は自動的に有理数体になります：

```
mat = matrix(2,2,[1/2,2,3,4])      # 行列の要素に有理数1/2がある
mat.base_ring()
```

実行結果の例

```
Rational Field
```

base ring をはじめから指定して行列を定義したい場合には，matrix の直後に書いて指定します。たとえば base ring を SR (symbolic ring=形式的な和・積・スカラー倍が定義された環) にしたい場合は次のようにします：

```
mat = matrix(SR, [[1,2],[3,4]])        # 行列をSR上で定義
mat.base_ring()
```

実行結果の例

```
Symbolic Ring
```

行列を定義したあとで base ring を変える場合は a を行列として a = matrix(SR, a) とします：

```
a = matrix([[1,2],[3,4]])      # 行列aを定義
print( a.base_ring() )         # 行列aのbase ringをプリント
a = matrix(SR, a)              # aのbase ringをSRに変更
print( a.base_ring() )         # 行列aのbase ringをプリント
```

実行結果の例

```
Integer Ring
Symbolic Ring
```

35.6 行列に対する命令

行列に対して実行できる演算には次のようなものがあります：

名前	Sage での記号	数学的意味	補足
固有多項式 (特性多項式)	A.charpoly()	$\det(xE - A)$	多項式の変数は x
固有値	A.eigenvalues()	$\det(\lambda E - A) = 0$ の解 λ	
固有ベクトル	A.eigenvectors_right()	$Av = \lambda v$ を満たす v	
最小多項式	A.minimal_polynomial()		多項式の変数は x
指数関数	A.exp()	e^A	
対角化	A.eigenmatrix_right	$P^{-1}AP$ は対角行列	P^-1AP と P の組

[*24] QQbar 同士の算術計算は厳密なのですが，QQbar を base ring とする行列の固有値の計算では近似計算になるようです

35.7　固有値・固有ベクトル

正方行列 A があるベクトル $v(\neq 0)$ と定数 λ に対して

$$Av = \lambda v$$

を満たすとき，λ を固有値，v を対応する固有ベクトルといいます。上の方程式を $(A-\lambda E)v = 0$ と書き直すと，これが非自明な解を持つための必要十分条件は $\det(A-\lambda E) = 0$ であることがわかります。つまり多項式 $\det(A-\lambda E) = 0$ の解が固有値です。先に述べたように，Sage で固有値をする場合，`base_ring` の設定によって答えの厳密さが異なります。それでは，具体的に行列

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

の固有値を計算してみましょう。この行列の固有値多項式は $f(\lambda) := (\lambda-1)(\lambda-4)-6$ なので $f(\lambda) = 0$ の解は $(5 \pm \sqrt{33})/2$ です。これを Sage で計算してみます。

```
1 mat = matrix(SR, 2,2,[1,2,3,4])     #base_ringはSRに設定
2 mat.eigenvalues()                   # 固有値のリストをプリント
```

実行結果の例

```
[-1/2*sqrt(33) + 5/2, 1/2*sqrt(33) + 5/2]
```

厳密な固有値が計算できました。しかし，固有多項式が具体的に解けない場合には，固有値は表示されません。たとえば，

```
1 mat5 = matrix(SR, 5,5, lambda i,j : 1/(i+j+1)) # 環をSRとして行列を定義
2 print(mat5)                                    # 行列をプリント
3 print(mat5.eigenvalues())                      # 固有値をプリント
```

と行列を定義してその固有値を計算させたとしても，実行結果は

実行結果の例

```
[  1 1/2 1/3 1/4 1/5]
[1/2 1/3 1/4 1/5 1/6]
[1/3 1/4 1/5 1/6 1/7]
[1/4 1/5 1/6 1/7 1/8]
[1/5 1/6 1/7 1/8 1/9]
Traceback (click to the left of this block for traceback)
...
ArithmeticError: could not determine eigenvalues exactly using symbolic
matrices; try using a different type of matrix via self.change_ring(),
if possible
```

となり，固有値を求められませんでした[*25]。この行列の固有値は 5 次方程式であり，一般解を具体的に表すことはできません。しかし，数値的には固有値を求めることはできます[*26]。

例えば上で作った 2×2 行列の係数体を CDF にして計算します：

```
1 mat = matrix(CDF,mat)     # base_ringをCDFに変更
2 mat.eigenvalues()
```

実行結果の例

```
[-0.372281323269, 5.37228132327]
```

固有値の数値のリストが表示されました。はじめから数値結果だけ得られればよい場合には SR を使わずに RDF や CDF を使いましょう。そのほうが SR で固有値を計算するよりもずっと高速に計算できます。

固有ベクトルは行列に対するメソッド `eigenvectors_right()` で求めます。

[*25] 5 × 5 行列であるところを 4 × 4 行列にすると厳密な固有値が得られます。
[*26] もちろん結果は近似値となりますが。

```
mat2 = matrix(SR, 2,2,[1,2,3,4])      # base_ringはSRに設定
eigs = mat2.eigenvectors_right()       # 固有ベクトルの組を定義
print(eigs)
```

実行結果の例

```
[(-1/2*sqrt(33) + 5/2, [(1, -1/4*sqrt(33) + 3/4)], 1), (1/2*sqrt(33) +
5/2, [(1, 1/4*sqrt(33) + 3/4)], 1)]
```

これは，次を意味しています：

[（第 1 固有値, [対応する固有ベクトル], 多重度），（第 2 固有値, [対応する固有ベクトル], 多重度）]

なので，少し面倒ですが，固有ベクトルを取り出すには，上のプログラムに続けて次のようにします。

```
print( eigv[0][1][0] )     # 一つ目の固有ベクトル
print( eigv[1][1][0] )     # 二つ目の固有ベクトル
```

実行結果の例

```
(1, -1/4*sqrt(33) + 3/4)
(1, 1/4*sqrt(33) + 3/4)
```

35.8 対角化とその応用

35.8.1 行列の対角化

行列の冪や指数関数を計算することは応用上重要です。それら計算するために必要な手順が行列の「対角化」です。行列 A を対角化するとは，ある正則行列 P を見つけて

$$P^{-1}AP = \mathrm{diag}[\lambda_1, \cdots, \lambda_n] : \text{対角行列}$$

と対角行列の形にすることです。対角化するための行列 P は A の固有ベクトル（縦ベクトル）を横に並べて作られます。

以下では，行列 $A = \left(\begin{smallmatrix} 1 & 4 \\ 2 & 3 \end{smallmatrix}\right)$ に対して，Sage で用意された命令 `eigenmatrix_right()` を用いて，対角化を計算しています。

```
A = matrix(SR, [[1,4],[2,3]] )      # 行列の定義
A.eigenmatrix_right()               # 固有値と対角化行列を求める
```

実行結果の例

```
(
[ 5  0]  [   1    1]
[ 0 -1], [   1 -1/2]
)
```

これは A の固有値が $5, -1$ で

$$P^{-1}AP = \begin{pmatrix} 5 & 0 \\ 0 & -1 \end{pmatrix}, \qquad P = \begin{pmatrix} 1 & 1 \\ 1 & -1/2 \end{pmatrix}$$

となることを意味しています。実際

$$\begin{pmatrix} 1 & 1 \\ 1 & -1/2 \end{pmatrix}^{-1} \begin{pmatrix} 1 & 4 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & -1/2 \end{pmatrix} = \begin{pmatrix} 1/3 & 2/3 \\ 2/3 & -2/3 \end{pmatrix} \begin{pmatrix} 1 & 4 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & -1/2 \end{pmatrix} = \begin{pmatrix} 5 & 0 \\ 0 & -1 \end{pmatrix}$$

となり，ちゃんと対角化されているのが分かります。

35.8.2 行列の指数関数と線形微分方程式

行列 A に対して，その指数関数 $\exp(tA)$ を

$$\exp(tA) = \sum_{n=0}^{\infty} \frac{t^n}{n!} A^n, \qquad t \in \mathbb{C} \tag{34}$$

によって定義します。Sage で行列の指数関数を計算することができますが，以下ではこれを応用して微分方程式を解きます。

最も簡単な例に対して，具体的な計算を行ってみましょう。ばねにつながれた質量 m からなる 1 次元の力学系を考えます。ばねのつり合いの位置を原点としましょう。このとき運動方程式から

$$m\frac{d^2x(t)}{dt^2} = -kx(t)$$

が成り立ちます。ここで k はバネ定数です。粒子は時刻 0 に位置 X にいて，速度 V であるとします。ここでの目標は時刻 t での位置 $x(t)$ を求めることです。この微分方程式は定数変化法などにより解くことができますが，いまは行列を用いて解いてみます。まず方程式を線形化します。$\sqrt{k/m} = w$ と置きます。$v(t) = dx(t)/dt$ とすると上の微分方程式は

$$\frac{d}{dt}\begin{pmatrix} x(t) \\ v(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ -w^2x(t) \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -w^2 & 0 \end{pmatrix}\begin{pmatrix} x(t) \\ v(t) \end{pmatrix}$$

と同値です。つまり

$$A = \begin{pmatrix} 0 & 1 \\ -w^2 & 0 \end{pmatrix}$$

と置けば，

$$\frac{d}{dt}\begin{pmatrix} x(t) \\ v(t) \end{pmatrix} = A\begin{pmatrix} x(t) \\ v(t) \end{pmatrix}$$

です，これはベクトル ${}^t(x(t), v(t))$ を一回微分するごとに行列 A が前に出てくることを意味しているので，$(x(0), v(0)) = (X, V)$ を思い出すと

$$\begin{pmatrix} x(t) \\ v(t) \end{pmatrix} = \exp(tA)\begin{pmatrix} X \\ V \end{pmatrix}$$

となります。微分方程式を解く問題は行列の指数関数 $\exp(tA)$ を求める問題に置き換わりました。Sage で行列 T の指数関数 $\exp(T)$ を求める命令が `T.exp()` です。具体的に Sage に行列を計算させて微分方程式を解いてみましょう：

```
var('t w')                          # t wを変数とする
A = matrix([[0,1],[-w^2,0]])        # 行列Aを定義する
B = t*A                             # BをtAとする
expo = B.exp()                      # expoをexp(B)とする
```

これで，$B = \exp(tA)$ が求まりました。つぎに

```
var("X V")              # X, Vを変数とする
y = vector([X,V])       # y をベクトル(X,V)とする
sol = expo*y            # sol をexp(tA) を(X,V)に作用させたものとする
```

とします。最後得られた量 `sol[0]` は $(x(t), v(t))$ なのでその最初の成分（第 0 成分）が時刻 t の位置を与えます。つまり微分方程式の解 $x(t)$ は `sol[0]` です。得られる関数は実なので実部を取って（これは何の変化ももたらしません）`full_simplify` を用いて整理します。上のプログラムに続けて

```
print( real_part(sol[0]).full_simplify() )
```

と入力すると最終的な答え

```
(X*w*cos(t*w) + V*sin(t*w))/w
```

が得られます。これが微分方程式の解 $x(t)$ です。通常の数学の記法で書けば，微分方程式の解は

$$x(t) = X\cos(wt) + \frac{V}{W}\sin(wt),$$
$$x(0) = X, \quad x'(0) = V$$

となるという事を意味しています。

35.8.3　行列の対角化とその応用：フィボナッチ数列の一般項

フィボナッチ数列 $a_1 = 1, a_2 = 1,$

$$a_{n+2} = a_{n+1} + a_n, \qquad n = 1, 2, 3, \cdots \tag{35}$$

を考えます。まず，上の関係式は

$$\begin{pmatrix} a_{n+2} \\ a_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a_{n+1} \\ a_n \end{pmatrix}$$

となる事に注意します。右辺の行列を A とします。すると

$$\begin{pmatrix} a_{n+1} \\ a_n \end{pmatrix} = A \begin{pmatrix} a_n \\ a_{n-1} \end{pmatrix} = A^2 \begin{pmatrix} a_{n-1} \\ a_{n-2} \end{pmatrix} = \cdots = A^{n-1} \begin{pmatrix} a_2 \\ a_1 \end{pmatrix} = A^{n-1} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \tag{36}$$

なので A^n が具体的に計算できれば，数列の一般項 a_n が求まる事になります。A^n を求めるために，A を対角化してみましょう。

```
A = matrix( SR, [ [1,1],[1,0] ])
sol = A.eigenmatrix_right()
P = sol[1]
D = P^(-1)*A*P
print('P = \n', P,  '\n')
print('D = P^(-1)AP =' )
print(D.expand())
```

実行結果の例

```
P =
[                   1                    1]
[-1/2*sqrt(5) - 1/2  1/2*sqrt(5) - 1/2]

D = P^(-1)AP =
[-1/2*sqrt(5) + 1/2                    0]
[                 0  1/2*sqrt(5) + 1/2]
```

ここで D を単純化するために expand() を行いました。この事から A を対角化する行列は

$$P = \begin{pmatrix} 1 & 1 \\ -(\sqrt{5}+1)/2 & (\sqrt{5}-1)/2 \end{pmatrix}$$

であり，$P^{-1}AP = \mathrm{diag}[(-\sqrt{5}+1)/2, (\sqrt{5}+1)/2]$ となることが分かります。この式の両辺を n 乗すると

$$\begin{pmatrix} (\frac{-1+\sqrt{5}}{2})^n & 0 \\ 0 & (\frac{\sqrt{5}+1}{2})^n \end{pmatrix} = P^{-1}A^nP$$

となります。したがって A^n は

$$A^n = P \begin{pmatrix} (\frac{-1+\sqrt{5}}{2})^n & 0 \\ 0 & (\frac{\sqrt{5}+1}{2})^n \end{pmatrix} P^{-1} \tag{37}$$

と計算する事ができます。これをつかって A^{n-1} を計算させてみましょう：

```
var('n')
Dn = matrix([ [D[0][0]^(n-1), 0], [0, D[1][1]^(n-1)]])    # Dのn-1乗を作る
An = P * Dn * P^(-1)                    # これがAのn-1乗
print expand(An)                        # Anを少し単純化してプリント
```

実行結果の例

```
[ 1/5*sqrt(5)*(1/2*sqrt(5) + 1/2)^n/(sqrt(5) + 1) +
1/5*sqrt(5)*(-1/2*sqrt(5) + 1/2)^n/(sqrt(5) - 1) + (1/2*sqrt(5) +
1/2)^n/(sqrt(5) + 1) - (-1/2*sqrt(5) + 1/2)^n/(sqrt(5) - 1)
2/5*sqrt(5)*(1/2*sqrt(5) + 1/2)^n/(sqrt(5) + 1) +
2/5*sqrt(5)*(-1/2*sqrt(5) + 1/2)^n/(sqrt(5) - 1)]
[
2/5*sqrt(5)*(1/2*sqrt(5) + 1/2)^n/(sqrt(5) + 1) +
2/5*sqrt(5)*(-1/2*sqrt(5) + 1/2)^n/(sqrt(5) - 1)
-1/5*sqrt(5)*(1/2*sqrt(5) + 1/2)^n/(sqrt(5) + 1) -
1/5*sqrt(5)*(-1/2*sqrt(5) + 1/2)^n/(sqrt(5) - 1) + (1/2*sqrt(5) +
1/2)^n/(sqrt(5) + 1) - (-1/2*sqrt(5) + 1/2)^n/(sqrt(5) - 1)]
```

D^{n-1}, A^{n-1} を Dn, An と定義しました。複雑ですが正しく計算できています。上の行列にベクトル

$$\begin{pmatrix}1\\1\end{pmatrix} \tag{38}$$

を掛けて得られるベクトルの第 2 成分が一般項 a_n となります。上のプログラムに続けて次を実行します。

```
assume(n,'integer')                     # nを整数と仮定する
v = vector([1,1])       # ベクトル(1,1)をvと定義
aa = An * v               # A^nをvに掛けると
aa[1].simplify_full()     # その第2成分が一般項となる
```

実行結果の例

```
-1/5*((sqrt(5) - 1)^n*(-1)^n - (sqrt(5) + 1)^n)*sqrt(5)/2^n
```

最後の結果が，フィボナッチ数列の一般項 a_n となります。これを整理すれば，

$$a_n = -\frac{\left(5^{\frac{1}{2}n}(-1)^n\left(\sqrt{5}-1\right)^n - \left(\sqrt{5}+5\right)^n\right)5^{-\frac{1}{2}n-\frac{1}{2}}}{2^n} = \frac{1}{\sqrt{5}}\left\{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right\}$$

となります。

実際，最後の出力の結果を n=1,2,... に対して表示させると

```
def fib(n):
    return -1/5*((sqrt(5) - 1)^n*(-1)^n - (sqrt(5) + 1)^n)*sqrt(5)/2^n

for k in range(1,20):
    print fib(k).simplify_full()
```

実行結果の例

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

となり，フィボナッチ数列に一致していることがわかります。

35.9 ジョルダン標準形

Sage でジョルダン標準形を求めるには jordan_form をつかいます。次の行列

$$\begin{pmatrix}3 & 2 & -3\\ -2 & -1 & 1\\ 5 & 3 & -5\end{pmatrix}$$

の Jordan 標準形を求めるには次のようにします：

```
A = matrix(QQ, [[3,2,-3],[-2,-1,1],[5,3,-5]])
A.jordan_form(transformation=True)
```

実行結果の例

```
(
[-1  1  0]  [-3  4  1]
[ 0 -1  1]  [-3 -2  0]
[ 0  0 -1], [-6  5  0]
)
```

これは,

$$P=\begin{pmatrix}-3&4&1\\-3&-2&0\\-6&5&0\end{pmatrix}\quad J=\begin{pmatrix}-1&1&0\\0&-1&1\\0&0&-1\end{pmatrix}=P^{-1}AP$$

となることを意味しています。A の Jordan 標準形が J で変換行列する為の行列が P です。

35.10 練習問題

35.10.1 大きな行列の固有値のグラフの表示

$N=20$ と実数 g に対して行列

$$Q^{+}(g):=\begin{pmatrix}1&\sqrt{1}g&0&0&0&0&0&0\\\sqrt{1}g&0&\sqrt{2}g&0&0&0&0&0\\0&\sqrt{2}g&3&\sqrt{3}g&0&0&0&0\\0&0&\sqrt{3}g&2&\sqrt{4}g&0&0&0\\0&0&0&\sqrt{4}g&5&\sqrt{5}g&0&0\\0&0&0&0&\sqrt{5}g&4&\ddots&\\0&0&0&0&0&\ddots&\ddots&\sqrt{N}g\\0&0&0&0&0&&\sqrt{N}g&N+(-1)^N\end{pmatrix}+g^2I$$

$$Q^{-}(g):=\begin{pmatrix}-1&g&0&0&0&0&0&0\\g&2&\sqrt{2}g&0&0&0&0&0\\0&\sqrt{2}g&1&\sqrt{3}g&0&0&0&0\\0&0&\sqrt{3}g&4&\sqrt{4}g&0&0&0\\0&0&0&\sqrt{4}g&3&\sqrt{5}g&0&0\\0&0&0&0&\sqrt{5}g&6&\ddots&\\0&0&0&0&0&\ddots&\ddots&\sqrt{N}g\\0&0&0&0&0&&\sqrt{N}g&N-(-1)^N\end{pmatrix}+g^2I$$

を定義する。ここで I は単位行列である。$Q^{\pm}(g)$ は 21×21 行列である。$Q^{\pm}(g)$ の固有値を小さい順からそれぞれ $\lambda_1^{\pm}(g),\lambda_2^{\pm}(g),\lambda_3^{\pm}(g),\cdots,\lambda_{N+1}^{\pm}(g)$ とする。固有値のリスト

$$\begin{aligned}L1&=(\lambda_1^{+}(j/30))_{j=-90}^{90},\\L2&=(\lambda_2^{+}(j/30))_{j=-90}^{90},\\L3&=(\lambda_3^{+}(j/30))_{j=-90}^{90},\\L4&=(\lambda_4^{+}(j/30))_{j=-90}^{90},\\L5&=(\lambda_5^{+}(j/30))_{j=-90}^{90},\\K1&=(\lambda_1^{-}(j/30))_{j=-90}^{90},\\K2&=(\lambda_2^{-}(j/30))_{j=-90}^{90},\\K3&=(\lambda_3^{-}(j/30))_{j=-90}^{90},\\K4&=(\lambda_4^{-}(j/30))_{j=-90}^{90},\\K5&=(\lambda_5^{-}(j/30))_{j=-90}^{90}\end{aligned}$$

を作り list_plot でそれぞれのリストをプロットせよ。グラフはオプション plotjoined=True により隣り合う点同士を線で繋ぐこと。すべてのグラフを重ねて一つのグラフにせよ（固有値ごとに色を変えておくとよい）[*27]。

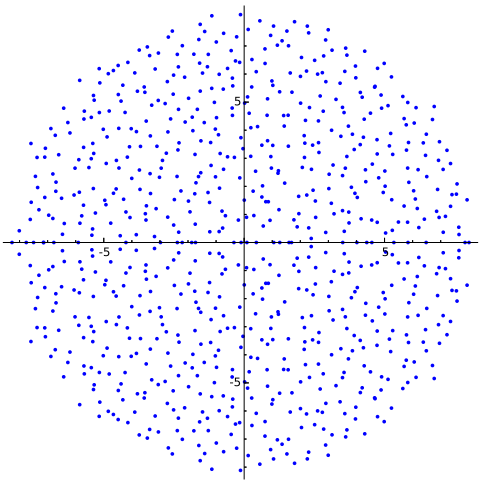
35.10.2 ランダム行列の固有値の分布

ランダムな数値を要素に持つ行列をランダム行列という。ランダム行列は，原子核の研究から現れたものであるが，最近になってその数学的研究は飛躍的に発展した。また，ランダムなポテンシャルを持つシュレディンガー作用素の固有値や固有ベクトルの挙動は不純物による絶縁体のメカニズムになったりする（アンダーソン局在）。

***

問題 区間 $(-1/2, 1/2)$ の中の一様な乱数を成分に持つ行列に持つ行列を作り，その固有値を複素平面上にプロットせよ。行列のサイズは 800×800 とし，縦横比は 1(aspect_ratio=1) とする[*28]。

そのような行列の固有値はある半径の円の中に均一に分布する事が知られている (circular law)。わかりやすい現象だが，これを数学的に証明するのは難しい。



ちなみに，Sage では random() で区間 $(0, 1)$ の中の数をランダムに返す：

```
for i in range(4):
    print(random())
```

実行結果の例

```
0.329694823321
0.831787904723
0.214856702912
0.615524329254
```

したがって，random()-1/2 で $(-1/2, 1/2)$ の一様な乱数を得ることができる。

```
for i in range(4):
    print(random() - 1/2)
```

[*27] 注：このような行列は 2 準位を持つ原子とレーザー光との相互作用を表すモデルのハミルトニアンを記述しており Rabi モデルと呼ばれる。対応する固有値はその量子系のエネルギー準位を表している。

[*28] 800×800 のサイズの行列の計算ではエラーが起こることがあるが，そのような場合には，400×400 のように行列のサイズを減らして計算してください。

実行結果の例

```
0.15646079682395098
0.006133788762921921
-0.2986405367263437
-0.06002764084916545
```

行列の base ring は何も指定しないか (その場合は RDF)，もしくは CDF にするとよい。グラフの描画は list_plot を使うのがよい。

36 乱数

乱数とはさいころの目のようにでたらめ[*29]に表れる数字のことです。乱数は，乱雑[*30]な現象を記述する場合だけでなく，決定的な量の近似値を計算するためにもしばしば用いられるます。ここでは Sage における乱数の基本的な使い方と簡単な応用を紹介します。以下のプログラムは Sage notebook で動作する事を確認していますが，`plot` 等の Sage の命令を除けば，純粋な Python プログラムとしても動作します。）

Python で使用される乱数はメルセンヌ・ツイスター (Mersenne twister) と呼ばれるアルゴリズムによって生成されます。これはある決まった手続きによって数を生成してゆくものですが，これは実用上十分なランダム性をもち非常に高速に生成する事ができるアルゴリズムです。ちなみにメルセンヌ・ツイスターは 1996 年頃日本人（松本眞氏・西村拓士氏）によって開発されました。これは巨大なメルセンヌ素数 $2^{19937}-1$ が素数であることを利用して作られるます。

コンピューターで作る乱数は，ある決定的な手続きで生成されます。したがって，それ自体にはパターンがあるので本当の意味での乱数ではありません。このようなものを擬似乱数といいます。どのような数列なら真の乱数といえるのかというのは難しい問題[*31]です。ひとつの擬似乱数は，ある種の問題を考えるときには，十分乱雑だけど，他の問題を考える場合には，乱雑さが不十分かもしれません。ここではどのようにして良い乱数を生成するのかという問題には触れずに，乱数の使い方と，その簡単な応用例を紹介したいと思います。

36.1 乱数の基本的な使い方

Python には `random` という標準ライブラリがあり，様々なタイプの乱数を生成することができます。

まずは次のプログラムを実行してみましょう：

```
1 import random     # ライブラリ random を呼び出す
2 random.seed(0)    # 乱数種を 0 とする
3 print "[0,1) の中の実数をランダムに生成する"
4 for i in range(5):
5       print random.random()      #[0,1) の実数をランダムにプリントする
```

実行結果の例

```
[0,1) の中の実数をランダムに生成する
0.844421851525
0.75795440294
0.420571580831
0.258916750293
0.511274721369
```

`random.seed(0)` は乱数の種の設定ですが，これについては後で説明します。`random.random()` で区間 $[0,1)$ の実数をランダムに作ることができます。[*32]

さて，上のプログラムをもう一度実行して見ましょう。実行結果は先ほどと全く同じになります。また隣の席の人とも同じになっているはずです。この数列は一見すると乱雑ですが，ある決定的な手続きで作られています。このようにある種の数学的な手続きで作られる数列だけど，一見するとパターンがなく乱数のように見えるものを疑似乱数といいます。

上の乱数には再現性がありますが，数値計算では乱数の再現性がない事にはあまり意味はなく，むしろ再現性のある方が有益な場合が多くあります。たとえば，ある数値計算をしていて予想外の結果が出たときに，それが偶然なのか必然なのかの原因を知りたいと思っても，同じ状況を再現できなければ，原因を突き止める事

[*29] 出鱈目
[*30] 乱雑の英語約は random（ランダム）だけど，二文字まで同じなのが不思議
[*31] 数学的なというより思想上の問題
[*32] $[0.0, 1.0)$ の中にある浮動小数点数が無作為に抽出されます。

が難しくなるでしょう[*33]。

乱数の列は乱数種 (random seed) ごとに決まっており，乱数種を変えることにより異なる乱数を使うことができます。例えば，上のプログラムで random.seed(1) のように乱数種 0 代わりに 1 を使えば，異なる乱数が得られます。乱数種を変えてみましょう。

```
1 import random
2 random.seed(1)
3 for i in range(5):
4     print random.random()
```

実行結果の例

```
0.134364244112
0.847433736937
0.763774618977
0.255069025739
0.495435087092
```

パスワードの生成などの時には，いつ実行しても同じ結果が出てしまうのは困るかもしれません。そういう場合には，そのときの時刻や乱雑なキー入力，マウスの動きから乱数を作ればよいでしょう。放射性物質の崩壊時間を利用すれば，真の乱数を作ることが出来ます[*34]。

プログラムを実行するたびに異なる乱数を使いたければ乱数種の指定を random.seed() とします。この場合，プログラムの実行時刻から乱数種が決まります：

```
1 import random
2 random.seed()   # 乱数種を時間から定める
3 print "[0,1)の中の実数をランダムに生成する"
4 for i in range(4):
5     print random.random()
```

実行結果の例

```
[0,1) の中の実数をランダムに生成する
0.0104079725999
0.605201486317
0.0631615144758
0.592872489434
```

このプログラムは毎回異なる乱数を生成します。このプログラムを何度か実行して結果が異なることを確認してみましょう。このプログラムの実行する時刻が全く同じでない限り，同じ結果は得られません。

$[0,1)$ の一様分布だけでなく，さまざまな乱数が用意されています。代表的なものを紹介します。

乱数の種類

- random() ： $[0,1)$ の中の実数をランダムに返す。
- randint(a,b) ：$a \leq N \leq b$ であるようなランダムな整数 N を返す
- choice(list) ： list の中からランダムに要素を取り出す。
- uniform(a,b) ：$[a,b]$ の中の実数をランダムに返す。
- gauss(mu,sigma) ：平均 mu，標準偏差 sigma のガウス分布に従うランダムな実数を返します。

乱数生成の命令はすべて random. で始まり，乱数の種類に応じてその後の命令を指定します。

たとえば random.choice(....) はライブラリの中の choice という命令を呼び出すことを意味します。たとえばリスト ['a','b','c'] からランダムに要素を取り出すには random.choice(['a','b','c']) のようにします。上記以外にも，多くの種類の乱数が用意されていますが，必要に応じて調べて下さい。

次のプログラムは 1 から 8 までの整数 1,2,3,4,5,6,7,8 をランダムに 20 個表示します：

[*33] 幽霊の目撃情報のように

[*34] 量子力学によれば，放射性物質の崩壊時間は真に確率的であり，それを正確に予測することは理論上不可能です

```
import random
random.seed(0)
for i in range(10):     # 1から8までの整数をランダムに抽出する
    print random.randint(1,8),
```

実行結果の例

```
7 7 4 3 5 4 7 3 4 5 8 5 3 7 5 3 8 8 7 8
```

次のプログラムではリスト x=["red", "green", "blue", "black", "white", "yellow"] の中から要素がランダムに抽出されます：

```
x = ["red", "green", "blue", "black", "white", "yellow"]
for i in range(10):           # リスト xの要素をランダムに10個とりだす
    print random.choice(x),
```

実行結果の例

```
green white yellow white blue red blue black yellow yellow
```

次は random.gauss(0,1) は平均 0，分散 1 のガウス分布に従う乱数です：

```
# 平均0，分散1の正規分布に従う乱数
for i in range(10):
     print random.gauss(0,1),
```

実行結果の例

```
-1.98153307955 0.288243745581 -0.119123311085 1.80432993192
-0.160362179057 -0.0506597136487 -0.190873889601 -0.990606238348
0.673029984025 -1.32408246447
```

一般に平均 μ，分散 σ^2 のガウス分布は

$$P(x) := \frac{1}{\sigma\sqrt{2\pi}} e^{(x-\mu)^2/(2\sigma^2)}$$

で定義され，実数値をとる乱数がガウス分布に従うとは，その乱数が $[a,b]$ に入る確率が

$$\int_a^b P(x)dx$$

である事と定義されます。この乱数が，本当にガウス分布であるかどうかは，ヒストグラムを書いてみるとよく分かります。

```
import random
random.seed(0)
lis = [random.gauss(0,1) for j in range(20000)] # ガウス分布に従う乱数のリスト
H = histogram( lis , bins=40, normed=True, color='linen')  # lisのヒストグラム
p(x) = 1/sqrt(2*pi)*e^(-x^2/2)    # gauss分布
P = plot( p(x), (x,-5,5) )        # p(x)をプロット
(H+P).show()
```

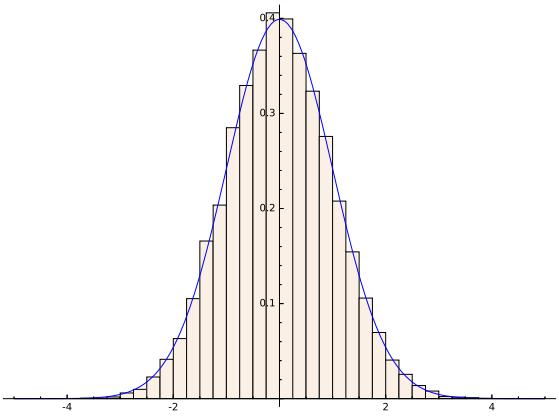


図 34　ガウス分布のヒストグラム

36.2　モンテカルロ法

36.2.1　円周率の近似

コンピューターにおける乱数の使い円周率を計算します。円周率を計算するアイディアは次の通りです：2 次元の領域 $[-1,1]\times[-1,1]$ に一様に n 個の点をばらまいて，半径 1 の円の中にある点の数を数えます。円の中の点の数を `count(n)` とするとき `4count(n)/n` が円周率を近似します。$[-1,1]\times[-1,1]$ の面積は 4 なので近似的に

$$(\text{ばらまいた点の数}) : (\text{円の中にある点の数}) \cong 4 : \text{円周率} \times (\text{半径} = 1)^2$$

なので

$$\text{円周率} \cong 4\frac{\text{円の中にある点の数}}{\text{ばらまいた点の数}}$$

となります。ばらまく点の数を無限に多くする極限で上式の右辺は円周率に収束します。

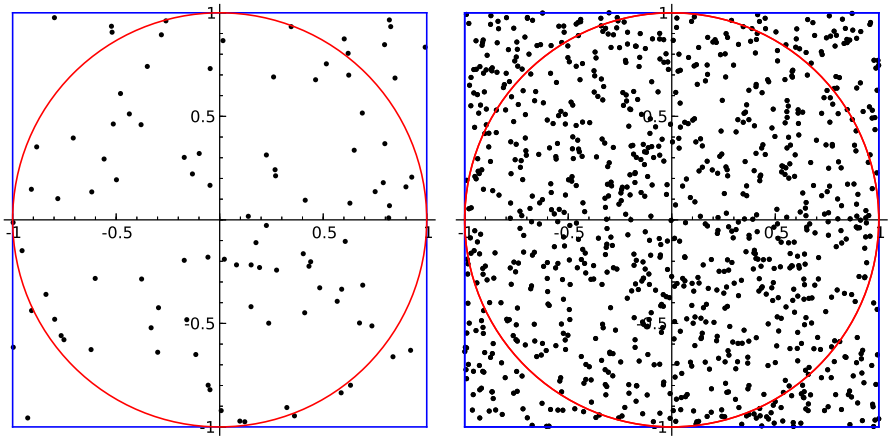


図 35　$[-1,1]^2$ に一様に点をばらまく

上のことをプログラムにすると次のようになります：

```
1 # モンテカルロ法で円周率を求める
2 import random
```

```
3  random.seed(0)
4
5  def count(n):
6      "n個の点を[-1,1]*[-1,1]にばらまいて，円の中にある点の個数を数える関数"
7      p=0     # 以下でカウントする値をpに入れる
8      for i in range(n):
9          x=random.uniform(-1,1)     # 乱数のx座標
10         y=random.uniform(-1,1)     # 乱数のy座標
11         if x^2+y^2<1:              # もしx^2+y^2<1ならば
12             p=p+1                      # pを一つ増やす
13     return p                       # カウントした点の数pを返す
14
15 n=100    # サンプル点の個数
16
17 pai = 4.0*count(n)/n     # これが円周率
18 print pai               #paiを出力する
```

実行結果の例

```
3.28000000000000
```

上のプログラムを実行してみましょう。出てきた結果が円周率の近似値です。サンプル点 n の値を 1 から 10000 ぐらいの範囲で変化させて，どのぐらい円周率の真の値 3.1415926535$\cdots$ に近くなるか試してみましょう。やってみると上のプログラムはそれほど正確な円周率の値を出さないことが分かります。

次に，上のプログラムで n について計算した円周率を pai(n) と定義して，これを n を変化させた結果を見てみましょう。

```
1  # ここまでは上のプログラム
2  # の13行目までと同じにする。
3
4  def pai(n):
5      return 4.0*count(n)/n
6
7  for i in range(1,51):
8      print 400*i, pai(400*i)
```

実行結果の例

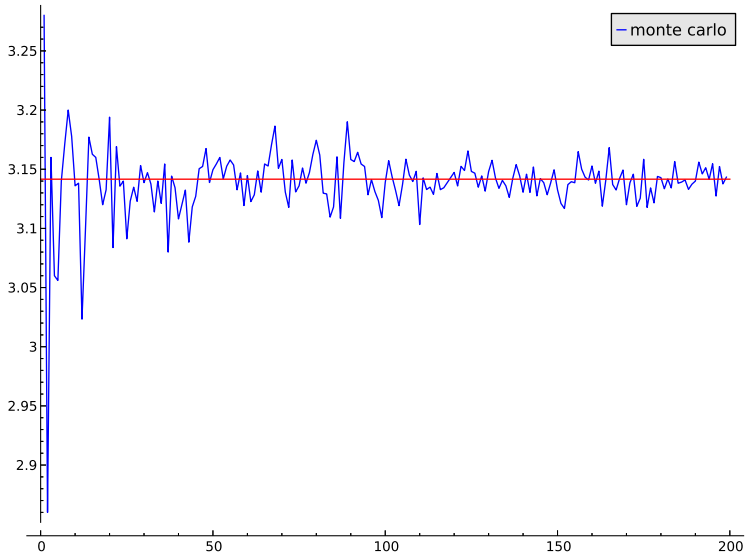
```
400 3.07000000000000
800 3.05000000000000
1200 3.14000000000000
......
19200 3.13895833333333
19600 3.11795918367347
20000 3.13840000000000
```

20000 個のサンプル点を取って計算してもそれほ 0.01 の桁が異なっています。これは確率論の基本的な事実で，分散を計算すればすぐにわかります。この方法ではサンプル点の数を n としたときに，円周率の誤差はおよそ $1/\sqrt{n}$ の割合になります。

さて，上のプログラムを改良して収束の様子をグラフで表してみましょう：

```
1  lis = []
2  for i in range(1,200):
3      lis.append([i,pai(100*i)])     # ペア[i,pi(100*i)]のリストを作る
4  p1 = list_plot(lis, plotjoined=True, legend_label='monte carlo')
5  p2 = line([(0,pi),(200,pi)], color='red')     # 円周率の真の値
6  (p1+p2).show()
```

計算には少し時間がかかりすぎる場合は上の `100*i` を `10*i` に変えてから実行してみましょう。次のようなグラフが表示されます：



36.2.2 収束の早さの比較

さて円周率は次のような交代級数で計算することもできます：

$$\pi = 4\arctan(1) = 4\int_0^1 \frac{1}{1+x^2}dx = 4\int_0^1 \sum_{n=0}^{\infty}(-1)^n x^{2n}dx = 4\sum_{n=0}^{\infty}(-1)^n\frac{1}{2n+1}$$
$$= 4\left(1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots\right)$$

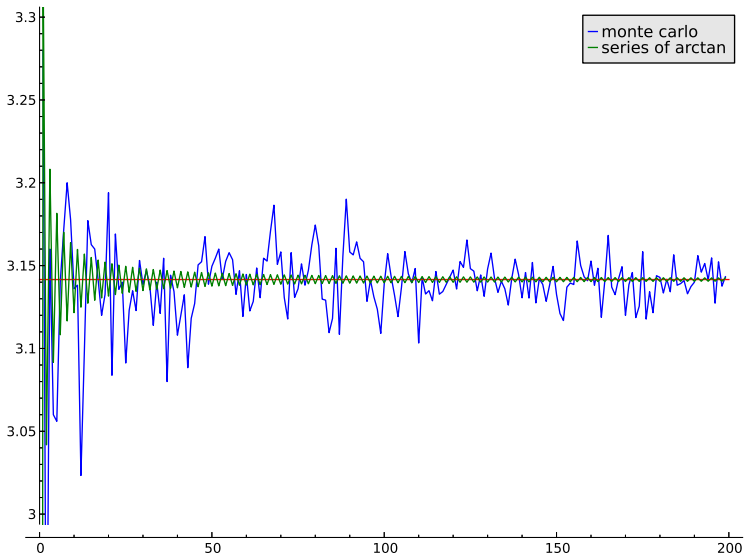
この級数の n 番目までの和の円周率の誤差は明らかに $1/n$ 程度です。このアルゴリズムを用いて計算して円周率と先ほどの乱数で計算した円周率との収束性を比較してみましょう：上のプログラムに続いて次を入力します：

```
# arctanの展開級数で求めた円周率の収束の様子をみる
def pi_arctan(n):
    "arctan(1)の級数展開のn個の和により近似した円周率"
    atan=0
    for j in range(n):
        atan=atan+(1.0/(2*j+1))*((-1)^j)
    return 4*atan

lis2=[]
for i in range(200):
    lis2.append((i,pi_arctan(5*i)))

p3 = list_plot(lis2, plotjoined=True, color='green',ymin=+3, ymax=3.3)
(p1+p2+p3).show()
```

arctan(1) の級数展開で計算した円周率の収束も必ずしも良いとは言えませんが，計算すればするほど，真の値に近づく事が分かります。一方モンテカルロ法は収束の速度は遅く，なかなか真の値に近づかない事が分



かります。モンテカルロ法で誤差 Δ の割合で結果を得るにはおよそ $1/\Delta^2$ 個のサンプル点を取る必要があります。モンテカルロ法が効力を発揮するのは高次元の積分値を求めるときです。

36.3　4 次元球の体積のモンテカルロ法による計算

4 次元球とは $\mathbb{R}^4$ の集合

$$B_4 = \{x = (x_1, x_2, x_3, x_4) \in \mathbb{R}^4 | x_1^2 + x_2^2 + x_3^2 + x_4^2 \leq 1\} \tag{39}$$

です。ここでは $[-1, 1]^4$ の中に点をばらまいて，B_4 の中にある点を数える事によって B_4 の体積を計算します：

$$(\text{ばらまいた点の数}) : (B_4 \text{ の中にある点の数}) \cong 2^4 : B_4\text{の体積}$$

なので

$$B_4\text{の体積} \cong 16 \times \frac{B_4 \text{ の中にある点の数}}{\text{ばらまいた点の数}}$$

です。したがって 4 次元球の体積を計算するためには次のようなプログラムを書けばよいでしょう：

```
1  ' モンテカルロ法で4次元単位球の体積を求める'
2  import random;    random.seed(0)
3  def count(n):
4      p=0     # カウントする数をpとする
5      for i in range(n):
6          x=random.uniform(-1,1)     # 乱数のx座標
7          y=random.uniform(-1,1)     # 乱数のy座標
8          z=random.uniform(-1,1)     # 乱数のz座標
9          w=random.uniform(-1,1)     # 乱数のw座標
10         if x^2+y^2+z^2+w^2 < 1:      # もしx^2+y^2+z^2+w^2<1ならば
11             p=p+1                  # pを一つ増やす
12     return p                       # カウントした点の数pを返す
13
14 n=2000    # サンプル点の個数
```

```
15 vol=16.0*count(n)/n      # これが体積
16 print vol                # volを出力する
```

上のプログラムを実行して出力を確認しましょう。 厳密な値は

$$|B_4| = \frac{\pi^2}{2} \cong 4.9348 \tag{40}$$

です。計算で得られた結果はこの値に近いでしょうか？

36.4　モンテカルロ法による積分

実関数 $f(x)$ に対して積分 $\int_0^1 f(x)dx$ を乱数を使って計算することを考えます。

まず，区間 $[0,1]$ に一様な乱数 X_j をばらまきます。このとき，X_j が区間 $(a,b) \subset [0,1]$ に入る確率は $b-a$ です。このとき

$$\int_0^1 f(x)dx = \frac{1}{N}\sum_{j=1}^{N} f(X_j) + \left(\frac{1}{\sqrt{N}}\text{程度の誤差}\right) \tag{41}$$

が成り立ちます[*35]。したがって，$N \to \infty$ とすれば

$$\int_0^1 f(x)dx = \lim_{N\to\infty}\frac{1}{N}\sum_{j=1}^{N} f(X_j) \tag{42}$$

となります。この事実を使って，積分 $\int_0^1 f(x)dx$ を近似することができます。積分区間が $[0,1]$ でないときには，適当にスケールを変えればよい。多重積分

$$\int_0^1 \cdots \int_0^1 f(x_1,\cdots,x_n)dx_1\cdots dx_n \tag{43}$$

を計算したいなら n 次元空間 $[0,1]^n$ に一様な乱数をばらまいてから和をとればよいということになります。つまり，$(X_j^1,\cdots,X_j^n)$, $j=1,\cdots,N$ を $[0,1]^n$ にばらまいた N 個の一様な乱数として上の多重積分を

$$\frac{1}{N}\sum_{j=1}^{N} f(X_j^1,\cdots,X_j^n) \tag{44}$$

によって近似します。この場合でも $1/\sqrt{N}$ のオーダーの誤差がともないます。

36.5　練習問題

$f(x,y,z) = \cos(x-y^2)e^{-x^2-y^2}/(x^2+z^2+1)$ とする。積分

$$\int_0^1 dx \int_0^2 dy \int_0^3 dz f(x,y,z) \tag{45}$$

をモンテカルロ法により計算するPython のプログラムを作成せよ。ただし，

- サンプル点の個数は 10000 個。
- 端末から実行したときに実行結果は積分の数値のみをプリントするようにする。
- ファイル名は `MonteCalroIntegral.py` とすること。

Python で $\cos(x)$ や e^x などの関数を使うときには，ファイルの先頭に

```
1 from math import *
```

と書いておけばよい。このとき $\cos(x)$, e^x はそれぞれ `cos(x)`, `exp(x)` で表す。

[*35] 重複大数の法則によれば誤差は $\log\log N/\sqrt{N}$ 程度だけど，$\log\log N$ は計算機で使用する範囲の N に対してはほぼ定数とおもってかまわない。

37　フーリエ解析

37.1　フーリエ変換

フーリエ級数については (9) で解説した。展開の式は $a_n = c_n + c_{-n}$, $b_n = i(c_n - c_{-n})$ とおくことによって次のように書かれる[*36]

$$f(x) = \sum_{n=-\infty}^{\infty} c_n e^{2\pi i n x/L}, \qquad x \in (0, L) \qquad \left(c_n = \frac{1}{L}\int_0^L f(x) e^{-2\pi i n x/L} dx \right).$$

この式で関数を $L/2$ だけ平行移動すれば

$$f(x) = \sum_{n=-\infty}^{\infty} c_n e^{2\pi i n x/L}, \qquad x \in (-L/2, L/2), \qquad \left(c_n = \frac{1}{L}\int_{-L/2}^{L/2} f(x) e^{-2\pi i n x/L} dx \right)$$

となる。

$f(x)$ は $\mathbb{R}$ 上の関数であるとする。上式を

$$f(x) = \sum_{n=-\infty}^{\infty} \frac{1}{L} \left(\int_{-L/2}^{L/2} f(y) e^{-2\pi i y \frac{n}{L}} dy \right) e^{2\pi i x \frac{n}{L}}, \qquad x \in (-L/2, L/2)$$

とし，$L \to \infty$ の極限をとると，適当な仮定のもとで和は積分となり

$$f(x) = \int_{-\infty}^{\infty} \left(\int_{-\infty}^{\infty} f(y) e^{-2\pi i k y} dy \right) e^{2\pi i k x} dk \tag{46}$$

となることがわかる。そこで，

$$\hat{f}(k) := \int_{-\infty}^{\infty} f(y) e^{-2\pi i k y} dy$$

を f のフーリエ変換という。関数 $g(k)$ に対して

$$\check{g}(x) := \int_{-\infty}^{\infty} g(k) e^{-2\pi i k x} dk$$

を g の逆フーリエ変換という。(46) は関数 f をフーリエ変換して逆フーリエ変換するともとに戻ることを意味している。この式をフーリエの反転公式という。また (46) は波数 k の波 $e^{2\pi i k x}$ の連続的な重ね合わせで関数 $f(x)$ を表現したといえるので，$f(x)$ に含まれる各振動数の振幅（スペクトル成分）が $\hat{f}(k)$ である。

Sage でいくつかの関数のフーリエ変換を計算してみよう。

- ガウス型関数のフーリエ変換

```
var('k')
f = exp(-x^2)
g = integral(f*exp(-2*pi*I*k*x), x,-oo,oo)
h = integral(g*exp(2*pi*I*k*x), k, -oo,oo)
print(f)
print(g)
print(h)
```

[*36] $e^{i\theta} = \cos\theta + i\sin\theta$ に注意

実行結果の例

```
e^(-x^2)
sqrt(pi)*e^(-pi^2*k^2)
e^(-x^2)
```

これは $f(x)=e^{-x^2}$ のフーリエ変換が

$$\hat{f}(k)=\int_{-\infty}^{\infty} f(x)e^{-2\pi ikx}dx=\sqrt{\pi}e^{-\pi^2k^2}$$

であり，その逆フーリエ変換は $f(x)$ であることを意味している。反転公式は $x^3e^{-x^2}$ などに対しても成り立つことを確かめることができる。

```
var('k')
f = x^3*exp(-x^2)
g = integral(f*exp(-2*pi*I*k*x), x,-oo,oo)
h = integral(g*exp(2*pi*I*k*x), k, -oo,oo)
print(f)
print(g)
print(h.simplify_full())
```

実行結果の例

```
x^3*e^(-x^2)
-1/2*(-2*I*pi^(7/2)*k^3 + 3*I*pi^(3/2)*k)*e^(-pi^2*k^2)
x^3*e^(-x^2)
```

反転公式は

$$\int_{-\infty}^{\infty}|f(x)|^2dx<\infty$$

である場合にほとんどの点 x で成り立つことが知られている。$f(x)=1/(1+x^2)$ などの場合にも成り立つのだが，フーリエ変換が k の正負で異なるため，Sage で計算させるのは面倒である。

37.2 離散フーリエ変換

時間間隔 T ごとにサンプルした数列

$$x(0),\ x(T),\ x(2T),\cdots,x((N-1)T)$$

が与えられたとする。このデータの離散フーリエ変換 $X(k)$ は

$$X(k):=\sum_{n=0}^{N-1}x(nT)e^{-2\pi ink/N},\qquad (k=0,1,2,\cdots,N-1)$$

で定義される。このとき，元データ x は X から逆変換

$$x(nT)=\frac{1}{N}\sum_{k=0}^{N-1}X(k)e^{2\pi ink/N},\qquad (n=0,1,2,\cdots,N-1)$$

によって復元することができる。

$T=1$ に対してリスト $[1,2,3,4]$ の離散フーリエ変換を計算するには次のようにする

```
xx = [1,2,3,4]
N = len(xx)

def f(k):
    return sum( [xx[n]*exp(-2*pi*I*n*k/N) for n in range(N)] )
```

```
6
7 XX = [f(k) for k in range(nn)]
8 print(XX)    #xxの離散フーリエ変換
```

実行結果の例

```
[10, 2*I - 2, -2]
```

つぎのプログラムでは与えられたに対してその離散フーリエ変換と逆離散フーリエ変換を返す関数を定義して計算を行っている。

```
1  def d_fourier(xx):
2      N = len(xx)
3      var('k')
4      f(k) = sum( [xx[n]*exp(-2*pi*I*n*k/N) for n in range(N)] )
5      XX = [f(k) for k in range(N)]
6      return XX
7
8  def d_inverse_fourier(XX):
9      N = len(XX)
10     var('k')
11     f(k) = sum( [XX[n]*exp(2*pi*I*n*k/N) for n in range(N)] )
12     xx = [(1/N)*f(k) for k in range(N)]
13     return xx
14
15 lis1 = [1,2,3,4]; print(lis1)
16 lis1 = d_fourier(lis1); print(lis1)
17 lis2 = d_inverse_fourier(lis1); print(lis2)
```

実行結果の例

```
[1, 2, 3, 4]
[10, 2*I - 2, -2, -2*I - 2]
[1, 2, 3, 4]
```

参考文献

[1] Python 公式ドキュメント https://docs.python.org/ja/3/

[2] Sage チュートリアル https://doc.sagemath.org/html/ja/tutorial/index.html

[3] Paul Zimmermann, Alexandre Casamayou, Nathann Cohen, Guillaume Connan, Thierry Dumont, Laurent Fousse, François Maltey, Matthias Meulien, Marc Mezzarobba, Clément Pernet, Nicolas M. Thiéry, Erik Bray, John Cremona, Marcelo Forets, Alexandru Ghitza, Hugh Thomas, *Mathematical Computation with SageMath*, https://www.sagemath.org/sagebook/english.html