Python と SageMath

佐々木格　(信州大学理学部)

2018 年 7 月 18 日

目次

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事が特徴です。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 は数学ソフトウェアを 100 個近く[*1]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，それらには互換性がありません。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。現在のところ[*2]，Sage のプログラムは Python2 の文法に従っていますので，以下では Python2 について解説を行います。

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] 正確には 90 個 2018 年 4 月現在

[*2] 2018 年 4 月 25 日現在の最新版は SageMath ver.8.1

第 I 部

Python の基礎

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 3 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
(3) Sage ノートブックから実行する

(3) については後半の Sage の解説で紹介します。

1.1　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：

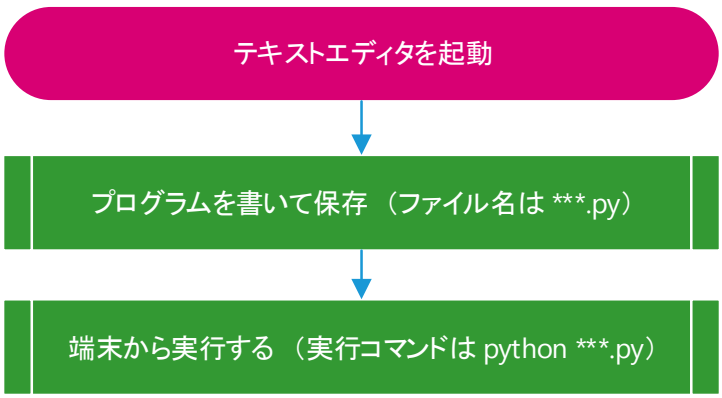


図 1　Python プログラムの実行方法 (1)

Python プログラムはテキストエディタで書きますが，以下では，テキストエディタとして Emacs を用いることを想定しています。次のようにファイル名 (`hello.py`) を付けてから Emacs を起動します。

```
1 user@debian:~$ cd ~/Desktop/dataproc1/              # ディレクトリを移動
2 user@debian:~/Desktop/dataproc1$ emacs hello.py &       # Emacsを起動
```

Emacs が起動したら `C-x C-s` を押してファイルを保存しましょう。これで，デスクトップにあるディレクトリ `dataproc1` にファイル `hello.py` が作られました。Emacs ウィンドウの下部のバーの表示が Python モードになっていることを確認してください。

さて，`hello.py` へ次の内容を書きましょう：

- ファイル名：**hello.py**

```
1 print 'Hello World'
```

入力したら保存 (`C-x C-s`) 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，端末のウィンドウに戻りましょう[*3]。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。

[*3] ウィンドウを切り替えは Alt+Tab で行い，極力マウスを使わないことをおすすめします。

```
user@debian:~/Desktop/dataproc1$ python hello.py
Hello World
user@debian:~/Desktop/dataproc1$
```

上のように端末に Hello World と表示されれば成功です。Python では，プログラムを実行する際に，C 言語のように事前にコンパイルをする必要はありません。

1.2 日本語を含む Python プログラム

日本語を含む Python プログラムは，文字コードを utf-8 で書き，ファイルの先頭に次の一文を追加しておかなければなりません。

```
# -*- coding: utf-8 -*-
```

もしくは

```
# coding: utf-8
```

Emacs を使って編集する場合はひとつ目の方を推奨します。この一文を入れることで，Python 実行時に，日本語が含まれていることが認識され，日本語の文字の処理が正常に行われるようになります。

例えば，つぎのようなプログラムを作成します：

- ファイル名：**hellojp.py**

```
print 'こんにちは'
```

端末から python hellojp.py とすることで実行すると

実行結果の例

```
sys:1: DeprecationWarning: Non-ASCII character 'xe3' in file test.py on line 1, but no
encoding declared; see http://www.python.org/peps/pep-0263.html for details
こんにちは
```

のようなエラーメッセージが出ます。これを解決するために，上で述べたようにプログラムの先頭にコーディングの指定をします：

- ファイル名：**hellojp.py**

```
# -*- coding: utf-8 -*-
print 'こんにちは'
```

上のように記述してから python hellojp.py とプログラムを実行すれば，次のように日本語を表示することができます：

実行結果の例

```
こんにちは
```

1.3 コメントアウト

Python のプログラムでは，コメントは『#』の後に書きます[*4]。コメントアウトされた部分はプログラムを実行する際には無視されます：

```
print 'Hello World'       # この部分は無視されます
```

[*4] 一方で LaTeX のコメントは % の後に書きます。

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，プログラムのファイルを作成せずに，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]` を実行するだけです。

```
user@debian:~$ python
Python 2.7.9 (default, Jun 29 2016, 13:08:31)
[GCC 4.9.2] on linux2
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    # 注意！商を返します！
2
>>> 9%4    # 割り算のあまり
1
>>> 9.0/4   # 小数点数での割り算
2.25        # 小数点以下も計算される
>>> 2**10   # 2の10乗
1024
```

Python2 では整数同士の割り算 (/) では余りを切り捨てるので注意が必要です。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 ではこの記法はビットごとの排他的論理和 (XOR) に割り当てられており，冪は 2^{10} は `2**10` のように表します。Python3 ではスラッシュ『/』で小数点以下も割り算がおこなわれるように変更されました。後半で解説する Sage のユーザーは数学者が多く，排他的論理和を多用する数学者は少ないので，^は冪を意味するように変更されています。また `5/3` は $\frac{5}{3}$ という有理数を表します。

インタラクティブシェルを終了するには `CRTL+d` もしくは `exit()` を入力します

```
>>> exit()                          # もしくは ctrl+d
user@debian:~$
```

2 変数

変数を使った計算を，インタラクティブシェルを使って紹介をします。

```
>>> 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 ?                        # エラーメッセージ
NameError: name 'b' is not defined                    # エラーメッセージ
>>> b = -4
>>> a+b
10
```

定義されていない変数を使おうとするとエラーメッセージが出ます。

変数名はある程度自由に決めることができますが，いくつかのルールがあります。変数名はアルファベット a–z, A–Z, から始めなければなりません。変数名では大文字と小文字は区別されます。先頭以外では，さらに数字 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。

3　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
```

4　文字列，数値，データの型

ここまでに，Python で変数，文字列，数を扱いました。数は 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。

4.1　文字列

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
>>> aa
'Alice'
>>> aa + 'Bob'
'AliceBob'
>>> aa + ' and Bob'
'Alice and Bob'
```

4.1.1 文字列の中でのコーテーションと改行

文字列を定義するときにはその文字を『" "』か『' '』で囲みます。文字列の中でコーテーションをし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'       # コーテーションマークを含む文字列
```

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
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『\』を使い，改行文字や，『\』，『"』，『'』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
- \t　⇝　タブ

4.2 データの方を調べる

文字列は str 型 (string) と呼ばれます。

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

4.3 数値の型

上で 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。そして Python では整数と浮動小数点数の演算は浮動小数点数として実行されます：

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

また整数型 int は-2147483647 から 2147483647 までしかサポートしていません (PC 環境にもよります)。それ以上になると長整数 long 型を使う必要があります。次のようにして，これを確かめてみましょう。

```
>>> dd = 2**62              # ddは2の63乗
>>> dd
4611686018427387904
>>> type(dd)
<type 'int'>                # これはint型だけど
>>> ee = 2**65              # eeは2の65乗
>>> ee
92233720368547758O8L        # 最後のLはLong型の印
>>> type(ee)
<type 'long'>               # これはlong型
```

なぜ，同じ整数を扱うのに int 型と long 型があるのでしょうか？自分が電卓を使って計算をすることを考えてみましょう。電卓の桁数の中で収まる計算なら，ボタンを数回押すだけで計算は直ちに終了しますが，電卓の桁数をはみ出るような計算の場合は，紙を用意して，数字を何桁かに分割して書いて，桁ごとにそれぞれ計算して，最後に繰り上がりに注意しながら和をとり，最後の答えを紙に書く，という作業をしなければなりません。コンピューターの内部でも同じことが起きていて，桁数の少ない数（といっても普通生活では出会わないような大きな数）の計算は用意された CPU の命令で高速で実行することができますが，ある桁数を超えた計算は，より遅くなる別の手順で行わなければなりません。

ちなみに Python3 では int 型とよばれる整数型しかありません。これは python2 の long 型と同じように振る舞います。Python3 は高速性を犠牲にして，整数についてより直感的にプログラムを書けることを重視したということでしょう。

4.4 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 ?                        # エラーメッセージ
TypeError: cannot concatenate 'str' and 'int' objects  # エラーメッセージ
```

上エラーを見ればわかるように str 型と int 型は足せません。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：

```
>>> 'Alice' + str(1999)      #1999を文字列'1999'にしてから'Alice'に付け加える
'Alice1999'
```

逆に，文字列になっている数字列から整数や小数点数に変換してみましょう：

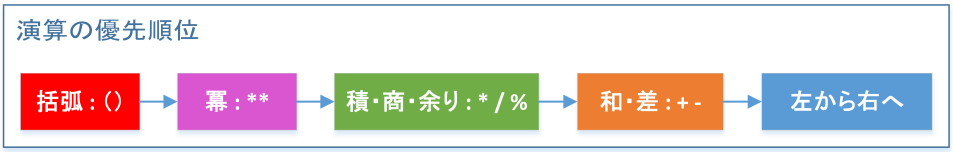
```
>>> aa = '123'       # '123'は文字列
>>> int(aa)
123                  # 123は整数
>>> float(aa)
123.0                # 123.0は浮動小数点数
```

5 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 (Addition)	10+20 は 30 を与える
-	差 (Subtraction)	20-10 は 10 を与える
*	積 (Multiplication)	4*5 は 20 を与える
/	除法 (Division)	14/3 は商 4 を与えるが 14.0/3 は $4.66\cdots67$ のように小数で計算される
%	余り (Modulus)	8%5 は余り 3 を与える
**	冪 (Power)	2**3 は $2^3 = 8$ を与える

括弧は例外なく，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*4/3**2 は次のような順で計算されます：

```
5*4/3**2 = 5*4/(3**2)          # 冪が最初に計算される
         = 5*4/9
         = (5*4)/9             # 左から計算される
         = 20/9
         = 2                   # 整数同士の割り算は，商が返される
```

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 = 7625597484987L
  (3**3)**3 = 19683
  3**(3**3) = 7625597484987L
```

この例では，冪は右から順に計算されています。したがって，冪（**）を複数回使用する場合は、括弧で囲んで順番を明確にすることをおすすめします。

Python での割り算『/』は整数同士では商を返すが，少数を含む計算では少数を返すので注意が必要です。上の計算例では

```
   5.0*4/3**2 = 5.0*4/9
              = 20.0/9
              = 2.2222222222222223
```

となります。

6　プリント (print)

インタラクティブシェルでは入力したものが順次ディスプレイに表示されますが，ファイルから Python を実行したときには print を書かない限り，画面には表示されません。

次のファイルを作って実行してみましょう：

- ファイル名：**print01.py**

```
a = 6
print 6
b = 4
a + b
```

実行結果の例

```
6
```

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はありません。print 文が複数回あるときは，改行されて表示されます：

```
print 4
print 5
```

実行結果の例

```
4
5
```

print 文の後に改行をさせたくない場合はコンマ『,』を付けます：

```
print 4,
print 2+3
```

実行結果の例

```
4 5
```

（注）上のように print 文の中で計算させることも出来ます。

7　リスト

リストとはデータのいくつかの集まりです。次を試してみましょう：

```
>>> a = ['Alice', 'Bob', 2,3,8]    # リストを定義してaに代入
>>> a                              # aの内容を確認
['Alice', 'Bob', 2,3,8]
>>> type(a)                        # aの型を確認
<type 'list'>                      # aの型はリスト
```

続いて，リストの成分を取り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
```

リストの成分の番号は 0 から始まることに注意しましょう。存在しない成分を呼び出すとエラーが出ます：

```
>>> 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，リストの要素を簡単に取り出すことが出来ます：

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

もっと大き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','i','j','k','l','m','n','o','p']
>>> print a[:5]
['a', 'b', 'c', 'd', 'e']
>>> print a[-5:]
['l', 'm', 'n', 'o', 'p']
```

次にリストを足し合わせるとどうなるか試してみましょう：

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

つまり，リストの和はリストを結合する事を意味します。リスト自身への要素の追加は append() という『メソッド』を使っても行うこともできます：

```
>>> 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出す
13
>>> aa
 ['a', 'b', 'c', 'd', 'e', 'b', 23, 8, 12]
>>> aa.pop(3)     #aaから3番目の要素を取り出す
'd'
>>> aa
['a', 'b', 'c', 'e', 'b', 23, 8]
>>> 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
```

連続する数字列のような特別なリストは最初から用意されています

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

8 Python で扱えるその他のデータ形式

8.1 タプル

リスト (list) のように要素を集めたものタプル (tuple) があります。タプル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
>>> print a
(3, 7, 'abc')
>>> type(a)
<type 'tuple'>      #aの型はタプル
```

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 です。

8.2 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}
```

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：

```
>>> print 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：

```
>>> b = a.keys()      #aの『鍵』の集まりをbとする
>>> print b
['death', 'type', 'name', 'birth']
>>> c = a.values()      #aの『値』の集まりをcとする
>>> print c
[1582, 'A', 'Nobunaga', 1534]
```

辞書から要素を削除するには，del を使って削除したいキーを指定します：

```
>>> del a['type']      # typeの鍵を削除
>>> 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
```

8.3 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。要素は 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]
>>> b = set(a)
>>> print b
set([1, 2, 3, 4])
>>> b.add(5)      #bに5を付け加える
```

```
>>>print(b)
set([1, 2, 3, 4, 5])
```

要素はソートされていて重複が除かれているのがわかります。

9　練習問題

9.1　問題：$e^{\pi} - 20$ の計算

pp $= 3.41592$，ee $= 2.718281$ とする。ee$^{\text{pp}} - 20$ 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 python problem01.py と実行したときに，答えを表示するようなプログラムでなければならない。

$e^{\pi} - 20$ 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。

10　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 raw_input() と input() という関数を使います。前者は文字列を取得するときに使い，後者は数値を取得するときに使います[*5]。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**

```
1 # -*- coding: utf-8 -*-
2
3 namae = raw_input('あなたは誰ですか？：')
4 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：

```
1 user@debian:~/Desktop/dataproc1$ python input1.py
2 あなたはだれですか？：
```

そこで，信州花子と入力して Enter キーを押せば

```
1 こんにちは　信州花子　さん
```

と出力されます。上のプログラムは次の手順で実行されました：

1. 「あなたはだれですか？：」と画面に表示
2. 変数 namae を作成。キーボード入力を待つ。
3. 入力された文字を変数 namae に代入
4. 「こんにちは・・・」を画面に表示

ここで重要な点は，raw_input() で入力された文字列は変数 namae に格納されることです。

入力するものが数値である場合には input() 関数を使います。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を作ってみます。

- ファイル名：**input2.py**

```
1 # -*- coding: utf-8 -*-
2
3 num = input('数値を入力してください：')
4 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：

```
1 user@debian:~/Desktop/dataproc1$ python input2.py
2 数値を入力してください：123
3 入力された数値の２乗は　15129　です
```

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 # -*- coding: utf-8 -*-
2
3 namae = raw_input('あなたの名前を入力してください：')
4 shintyo = input('あなたの身長は何センチですか？：')
```

[*5] Python3 では raw_input() は廃止されて，input() に置き換わりました。

```
5 weight   = input('あなたの体重は何キログラムですか？：')
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。

11　論理型と比較演算子

数学では，正しい命題は真，間違った命題は偽であるといいます。Python でも真偽値というものがあり，条件が正しいかどうかで真・偽の値が決まります。これらは条件に応じて処理を変化させるときに使います。

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『==，>，!=』という記号を用いました。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，それぞれ伝統的な数学記号 $=$，$\neq$，$<$，$>$，$\leq$，$\geq$ 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
```

データが与えられたときに，それらの関係に対して真偽値を得ることが出来ます。

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
```

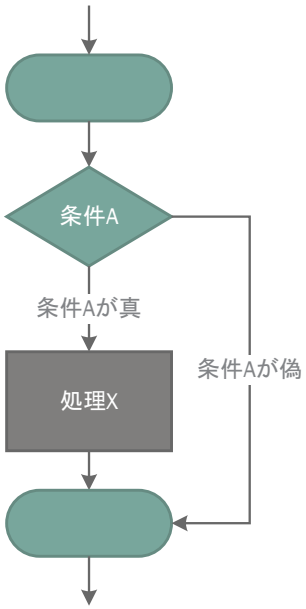
次のようにして，型を調べる type と組み合わせて使うことも出来ます：

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

12　条件分岐

12.1　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：



これを Python では次の文法で書きます：

Python での if 文の構造

```
1 if 条件A:
2     [ 処理X ]
3     [ 処理X ]
4     [ 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 数行にわたる場合は，上のようにインデントを揃える。

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で見えないのですが，インデントが揃った部分が，処理されるプログラムの塊（ブロック）であると判断されます：

if 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 Python プログラムにします。

(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。

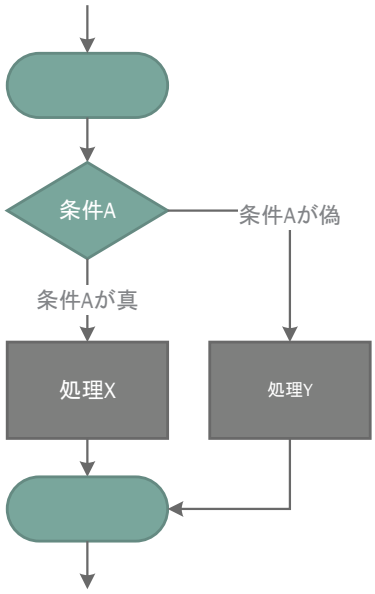
- ファイル名：**if1.py**

```
1 # -*- coding: utf-8 -*-
2 a = input('Input integer a: ')
3
4 if a%2 == 0:          # もしaが偶数なら
5     print 'a is even'
6     a = a/2           # aを半分にする。行頭のインデントは重要！！！
7
8 print a               # aの値をプリント
```

上のプログラムの 4〜6 行目が if 文です。実行していくつかの数値を入れて動作を試してみましょう。

12.2 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，より明示的に書くこともできます。

Python での if-else 文の構造

```
1 if 条件A:
2     処理X
3 else:
4     処理Y
```

- 条件 A が `True` であれば，処理 X が行い，`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) もし `a` が奇数ならば，`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**

```
1 # -*- coding: utf-8 -*-
2 a = input('a?:')
3
4 if a%2 == 0:      # もしaが偶数なら
5     a = a/2       #aを半分にする
6 else:             # そうでなければ
```

```
7     a = 3*a + 1   #aを3a+1にする
8
9  print a          #aの値をプリント
```

このプログラムを実行していくつかの数値を入れて動作を試してみましょう。

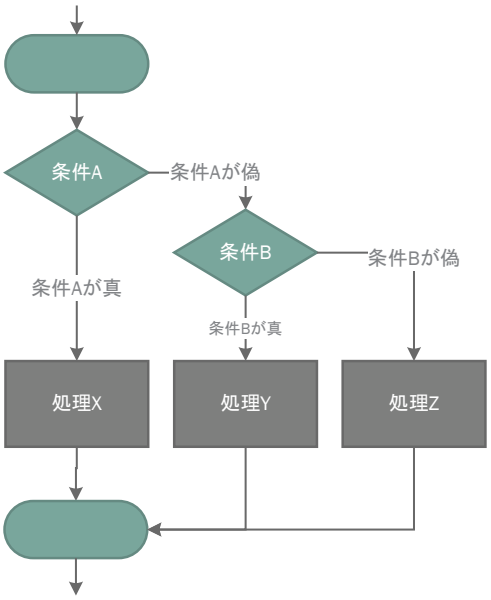
12.3 if-elif-else 文

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

```
1  if 条件A:
2      処理X
3  elif 条件B:
4      処理Y
5  else:
6      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 True 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。

12.4 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』が多くあって，論理構造がわかりにくく，そのままではプログラムにはしづらい。そこで閏年の条件を次のように書き直します：

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**

```
# -*- coding: utf-8 -*-
year = input('西暦を入力:')    # 入力した数値を変数yearに代入する

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。'
```

13　練習問題

13.1　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

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

- ファイル名：**bmi2.py**

```
# -*- coding: utf-8 -*-

namae = raw_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？：')
weight  = input('あなたの体重は何キログラムですか？：')

bmi = 10000.0*weight/(shintyo**2)
```

```
8  bmi = round(bmi,1)     #bmiの値を小数点以下2桁目を四捨五入する
9
10 print namae, 'さんのＢＭＩは', bmi, 'で',
11
12 if **********:
13     ***** 'やせ気味です。'
14 elif **********:
15     ***** 'ふつうです。'
16 **************:
17     ***** '太り気味です。'
18 ****:
19     ***** '太りすぎです。'
```

14　For 文による繰り返し

14.1　For 文の文法

同じ処理の繰り返しを記述するには for 文を使います。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，このような処理をより簡潔に書くことができます：

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

実際に，Python のインタラクティブシェルから上のプログラムを書けば次のような表示になるでしょう：

実行結果の例

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

ここで，range(5) はリスト [0,1,2,3,4] だったことを思い出しましょう[*6]：

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

上の，for 文で『for j in range(5):』は j を 0 から 4 まで順番に変えて，その下のブロックの処理を繰り返しているので，次のように j そのもの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
5
```

ここで j はダミー変数なので，他の文字に変えても結果全く同じです。例えば，次のプログラムは上のものと同じです：

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

[*6] Python3.x では range はリストを返しません。list(range(5)) がリスト [0,1,2,3,4] になります。

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>
```

- 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ません。これは for 文が終わってから実行されます。

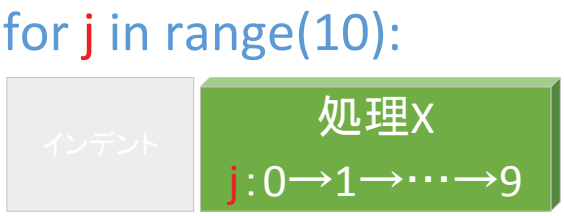


図 2　for 文の構造

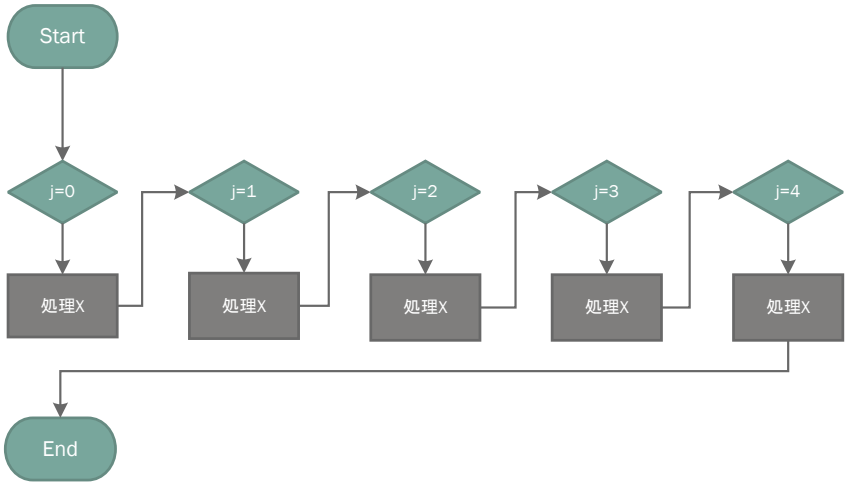


図 3　for 文で行われる処理 (5 回の繰り返し)

14.2　for 文のいくつかの例

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',         # 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',         # 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 # -*- coding: utf-8 -*-
2 for j in range(1,101):
3     print 'ひつじが'+str(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，その要素について順番に同じ作業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,          # iをプリントする
```

実行結果の例

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

14.3 for 文のインデントのエラー

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不当に下げた場合も同じエラーとなります：

- 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
```

14.4 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 1 b 1 c 2 a 2 b 2 c 3 a 3 b 3 c 4 a 4 b 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,                # 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
```

14.5 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
```

14.6 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 が現れたらその for 文から抜けます。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してゆきます，
- これは非常な早さで増加するので，もし 2^{2^j} の値が 100000000 を超えてしまった場合にはそこで for 文処理をやめたい。

- 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!!
```

15　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。

15.1　ファイルの書き込み (その 1)

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 プログラムで書きたいものを print しておけば，

```
user@debian :~/Desktop/dataproc1/06$
```

のように，端末でプログラムの置いてあるディレクトリで

```
$ python ファイル名.py > 出力するファイル名.txt
```

とすることによりプリントされたものをファイルに書き出すことができます。たとえば

```
$ python for4.py > for4result.txt
```

すると，フォルダには for4result.txt というファイルが新たに作られ，for4.py の実行結果が書き込まれています。ファイルの末尾に『追記』するには，『>>』を使います。試しに

```
$ python for3.py >> for4result.txt
```

を実行してみましょう。これで，for4result.txt のファイルの最後に，for3.py の実行結果が書き込まれました。

15.2　ファイルの書き込み（その 2）

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**

```
# -*- coding: utf-8 -*-
abc = 'hogehoge'                # 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
```

```
$ python write1.py
```

を実行すると test.txt が作られて hogehoge という文字列が書き込まれました。もう一度実行すると，文字列がさらに追加されます。追記ではなく，今あるものを消去してから書き込むには上の 3 行目の代わりに

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

とすればよいです。

例として，ひつじが・・・のプログラムをこの手順でファイルに書き出すことをやってみましょう。そのためには，まず文字列を作成しなければなりません。

- ファイル名：**CountSheep.py**

```
# -*- coding: utf-8 -*-

abc = ''        # 変数abcを空の文字列とする
```

```
4
5  for i in range(10):
6      abc = abc + 'ひつじが'+str(i)+'匹\n'    # \nは改行を意味する
7
8  f = open('hitsuji.txt', 'w')     # ファイル(hitsuji.txt)をwriteモードで開く
9  f.write(abc)
10 f.close()
```

このファイルを実行すれば，自動的に hitsuji.txt というファイルが作られ，中に，『ひつじが・・・』と書き込まれます。

15.3　ファイルの読み込み

ファイルを読み込むには read や readlines を使います。前者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，後者はファイルの各行のリストを作り出します。ここで，読み込むファイルを作るために次を実行します：

```
1 $ python for11.py > hoge.txt
```

ディレクトリには hoge.txt が生成されて，for11.py の実行結果が記録されました。さて，このファイルを read で読み込みます：

- ファイル名：**read1.py**

```
1  # -*- coding: utf-8 -*-
2
3  f = open('hoge.txt', 'r')  # 読み込みモードで開く
4  aaa =  f.read()            # hoge.txtの内容をaaaとする
5  print aaa                  # aaaの内容をプリントする
6
7  f.close()                  # hoge.txtを閉じる
```

実行結果の例

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

つぎに，各行を readlines で読み込みます：

- ファイル名：**read2.py**

```
1  # -*- coding: utf-8 -*-
2
3  f = open('hoge.txt', 'r')  # 読み込みモードで開く
4  aaa =  f.readlines()       # hoge.txtの各行からなるリストをaaaとする
5  print aaa                  # aaaの内容をプリントする
6
7  f.close()                  # hoge.txtを閉じる
```

実行結果の例

```
['\n','4\n','16\n','256\n','65536\n','4294967296\n','Too big!!\n',]
```

aaa は hoge.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。

16　練習問題

16.1　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
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：**collatz.py**

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

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

プログラムが完成したら実行して a として 19 を入力してみましょう。

実行結果の例

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

のようになればおそらく正しいプログラムです。

（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，実際に何回繰り返したらよいのか分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。

16.2 数字当てゲーム

1 から 1000 までの数字 x をランダムに生成して，x を 8 回以内で当てるゲームを作りたい。

乱数はライブラリをインポートすることで手軽に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。

```
1 import random    # 乱数のライブラリrandomをインポートする
2 x = random.randint(1,10)    # 1から10までの数字をランダムに生成しxとする。
3 print x
```

さて，ここでは，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 8 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 をランダムに生成されました。x を 8 回以内で当ててみよう！』と表示する。
(4) for 文で j を 0 から 7 まで変えながら，次の (5)–(8) の処理を繰り返す。
(5) `input` で『残り $8 - j$ 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，
(7) もし $a < x$ なら『もっと大きいよ』と表示し，
(8) そうでないなら『もっと小さいよ』と表示する。
(9) もし，$x \neq a$ なら次の (10),(11) を行う。
(10)『正解は x でした』と表示する。
(11)『残念 Game Over』と表示する。

上のアルゴリズムを Python で書きましょう。ファイル名は `find_number.py` とすること。

17　while 文による繰り返し

上の問題は $3n+1$ 問題に関するものでした。そこでは自然数 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 $n=1$ に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 $n=1$ という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 文によって記述しなければなりません。

17.1　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 を繰り返すプログラムです。
- まず条件 A が `False` なら処理 X は行われず while 文は終わります。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 文は終わりです。この繰り返しは，条件 A が `True` である限りずっと続きます。そして条件 A が `False` になったら while 文は終わり，その次の処理に進みます。
- オプションとして `break`, `continue` を使うとより柔軟な制御が可能になります：
  - 処理 X の実行中に `break` が現れると強制的に while 文から抜けます。
  - 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ジャンプします。

図 4　while 文の文法

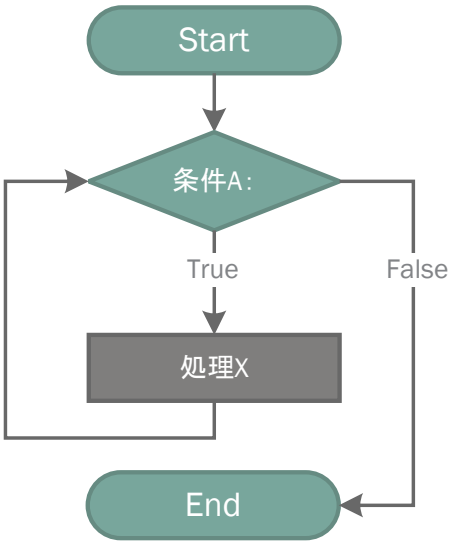


図 5　while 文の処理の流れ

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未満なら以下を続ける
    print a*a,     # aの二乗をプリントする
    a = a+1        # aの値を一つ増やす

print 'owari'      # while文を抜けてからこの処理を行う
```

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『print a*a; a = a+1,』です。while1.py では次のような処理を行っています：

1. 変数 a を 1 にセットする。
2. a<10 か調べる。a=1 だからは条件 a<10 は真なので
3. a*a=1 をプリントする。a の値を 1 増やす (a=2 になる)
4. a<10 を調べる。条件 a<10 は真なので
5. a*a=4 をプリントする。a の値を 1 増やす (a=3 になる)
6. ⋮
7. a*a=9 をプリントする。a の値を 1 増やす (a=10 になる)
8. a<10 を調べる。条件 a<10 は偽なので while 文の処理は終わる。
9. owari と表示する。

上の処理のように，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。次の for 文は上のプログラム while1.py と同じ出力になります。

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

print 'owari'
```

17.2 while 文の例

17.2.1 $3n+1$ 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 = input('Input an integer: ')    # 数を入力させて，その数をaに入れる
print a,       # 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,
```

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
```

17.2.2 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 # -*- coding: utf-8 -*-
2 j=1
3 while True:      # 条件は永遠に真なので，以下をずっと繰り返す！
4     print 'ひつじが'+str(j)+'匹'
5     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。

プログラムを終了するには CTRL + C と入力します。

17.2.3 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 # -*- coding:utf-8 -*-
2 j=1
3 while True:
4     print 'ひつじが'+str(j)+'匹'
5     j=j+1
6     if j > 123456:  # もし羊の数の二乗が123456を超えたらwhile文を終える
7         break
8
9 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**

```
a=0
while a<5:
    a = a+1
    print a,
else:
    print 'hogehoge'

print 'END'
```

実行結果の例

```
1 2 3 4 5 hogehoge 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 20 匹まで数えることにしましょう。

- ファイル名：**while_continue.py**

```
# -*- coding: utf-8 -*-
a=0
while a<20:
    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
羊が 16 匹
羊が 17 匹
羊が 18 匹
羊が 19 匹
羊が 20 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：

```
# -*- coding: utf-8 -*-
for i in range(1,21):
    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...'
```

17.2.4 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，どの n に対して a_n が 100000000 を超えるか分からないので while 文を使うのが便利です。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 とおく。
2. a=1 とおく。b=1 とおく。
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**

```
maxvalue = 100000000    # maxvalueを右辺の数字にセットする
a = 1      # a = 1 とおく
b = 1      # b = 1 とおく
while a < maxvalue:
    print a            # ここが
    a, b = b, a+b      # 繰り返されるブロック

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
```

17.2.5 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**

```
# -*- coding:utf-8 -*-
n = input('Input an integer(>1): ')    # 数を入力させて，その数を変数nに入れる
i = 2                  # i = 2 とおく
while i*i <= n:        # i*i≦nなら以下を繰り返す
    if n % i == 0:     # もしnがiで割り切れたら
        print n, 'is not a prime.'  # nは素数ではないとプリント
        break                       # while文抜けて12行目以下へ進む
    else:
        i += 1           # iの値を一つ増やす
else:                          # i*i>n となったら
    print n, 'is prime.'    # nは素数である

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。

18　練習問題

18.1　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なプログラムを作りなさい。

18.2　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。

- $b = 0$ なら a が最大公約数。
- $b \neq 0$ 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. a, b の値を入力させる。
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 = input('Input an integer: ')
b = input('Input an integer: ')
while *******:
    **********
    **********
    **********

print '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。

19　リストの操作とリスト内包表記

いろいろなデータをまとめて取り扱うときに便利なのがリストです。リストとは [3,6,3,9] のように，いくつかの要素を括弧の中に並べたものです。短いリストであれば，次のように直接的に書くことで定義します。

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

さらに，リストに要素を付け加えるには，append メソッドを使います。

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

もう少し多様な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。次の例を見てみましょう。

```
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]
```

上では，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。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')]
```

20　関数

あるまとまった処理を単純に繰り返すときには 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，簡単に使い回しできるようにしたものです。関数を使うことで一回書いたコードを再利用でき，スマートで読みやすいプログラムを作ることができるようになります。

図 6　関数でプログラムを再利用

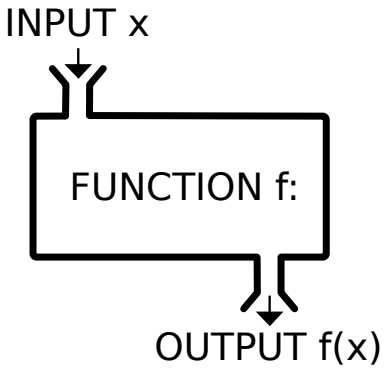


図 7　関数のイメージ

20.1　関数の定義のしかた

関数の定義

```
def 関数名(引数):
    [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]                # 戻り値が必要ない場合は省略可能
```

- 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。
- 関数名の大文字と小文字は区別されます。
- 引数や戻り値は不要なら省略できます。

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

print 'How are you?'         # How are you?と表示する
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。関数は定義された時点では実行されず，呼びだされたときに動きます。この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**

```
def sanjou(a):       # 関数名はsanjou, 引数はa
   return a*a*a      #aの3乗を返す

print sanjou(2)     # 返された8をプリントする
print sanjou(3)     # 返された27をプリントする
b = sanjou(1)+sanjou(2)+sanjou(3)+sanjou(4)
print b
```

実行結果の例

```
8
27
100
```

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**

```
def guuki(a):
  if a%2 == 0:         # もしaが偶数なら
     return 'even'     # evenを返す
  else:                # そうでなければ
     return 'odd'      # oddを返す

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**

```
def uruuQ(n):
    if n%400 == 0:
       return True
    elif n%100 == 0:
       return False
    elif n%4 == 0:
       return True
    else:
       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,        # 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 # -*- coding: utf-8 -*-
2 def CountSheeps(a):
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
5
6 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
```

関数を別の変数に代入することも出来ます

```
1 def nobu():
2     print 'De aruka'
3
4 oda = nobu        # 関数nobuを新たな変数odaに代入
5 oda()             # odaを実行
```

実行結果の例

```
De aruka
```

20.2 練習問題

20.2.1 最大公約数を計算する関数

18.2 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     **************
3         *******
4         *****
5         *****
6     ********
7
8 print gcd(23954187074819,8326543)
```

実行結果の例

```
32399
```

20.2.2 素数のリストを作成する

与えられた数 n に対して，n が素数なら `True` を返し，そうでなければ `False` を返す関数 `primeQ(n)` を定義せよ。そしてリスト内包表記によって 10000 以下の素数からなるリストを作成し，プリントするプログラムを作成せよ。ファイル名は `primelist.py` とすること。

20.3 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
```

20.4 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**

```
def daikei(a,b,h):    #a上底， b下底， h高さ
```

[*7] ただし，他で使われていないもの

```
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    #cはローカル変数なのでここでは定義されていないのでエラーとなる
```

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
```

20.5　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，最大公約数を求める関数 gcd(a,b) を定義するときには，次のようにコメントを書きます：

```
1 # -*- coding: utf-8 -*-
2
3 def gcd(a,b):
4     """ aとbの最大公約数を求める関数
5         ユークリッドの互除法により最大公約数を計算する
6     """
7     while b != 0:
8         a, b = b, a%b
9     return a
10
11 print gcd(1428,2618)
```

実行結果の例

```
238
```

上では 4,5,6 行目が関数の説明ですが，単なる文字列なのでプログラム実行時には無視されます。

20.6　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 # -*- coding:utf-8 -*-
```

```
def primeQ(n):
    """ nが素数ならTrue, 合成数ならFalseを返す関数を定義 """
    if n < 2:
        return False
    else:
        i=2
        while i**2 <= n:
            if n%i == 0:    #nがiで割り切れるなら
                return False
            i=i+1
    return True

# リスト内包表記をつかって, 1から100までの素数を列挙する。
print [k for k in range(1,101) if primeQ(k)]
```

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判定する関数 `is_prime` がデフォルトで用意されています。

20.7 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),        #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)      # ここで自分自身の関数kaijou(n-1)呼び出す！！

for i in range(10):               #i=0,1,2,…,9に対して
    print kaijou(i),              #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
```

ユークリッドの互除法はある種の手続きの繰り返しであることを利用して，最大公約数 gcd(a,b) を以下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) を実行する。
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) を実行する。
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。

このようにして，関数が繰り返しの動作によって定義可能な場合は，再帰的な定義によって非常に簡潔に関数を定義することが出来ます。

20.8 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。

(3), (4) に従って定義する関数 ${}_mC_n$ をそれぞれ `C(m,n)`, `D(m,n)` としてプログラムを書いてみましょう：

- ファイル名：**combinat1.py**

```
def kaijou(n):    # kaijou(n)=n!を定義
    if n == 0 or n==1:
        return 1
    else:
        return n*kaijou(n-1)

def C(m,n):     # (1)で組み合わせ数を定義
    return kaijou(m)/(kaijou(m-n)*kaijou(n))

def D(m,n):     # (2)で組み合わせ数を再帰的に定義
    if n<0 or m<n:
        return 0
    elif m == 0 or m==n:
        return 1
    else:
        return D(m-1,n) + D(m-1,n-1)

for i in range(7):     #i=0,1,2,3,4,5,6に対して
    print C(6,i),       #C(6,i)を表示

print ''    # 改行する

for i in range(7):      #i=0,1,2,3,4,5,6に対して
    print D(6,i),      #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になります。でも計算時間は大幅に異なります。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**

```
# -*- coding:utf-8 -*-
from time import time    #timeというモジュールからtimeという関数をインポート

def kaijou(n):
    if n == 0 or n==1:
        return 1
    else:
```

```
8         return n*kaijou(n-1)
9
10  def C(m,n):     # (1)で組み合わせ数を定義
11      return kaijou(m)/(kaijou(m-n)*kaijou(n))
12
13  def D(m,n):     # (2)で組み合わせ数を再帰的に定義
14      if n<0 or m<n:
15          return 0
16      elif m == 0 or m==n:
17          return 1
18      else:
19          return D(m-1,n) + D(m-1,n-1)
20
21  time1 = time()  # この時間をtime1とする
22  print C(23,12)    # C(23,12)を計算
23  time2 = time()    # このときの時間をtime2とする
24  print '計算に要した時間は', time2-time1, '秒'
25  print D(23,12)    # D(23,12)を計算
26  time3 = time()    # このときの時間をtime3とする
27  print '計算に要した時間は', time3-time2, '秒'
```

実行結果の例

```
1352078
計算要した時間は 0.000201940536499 秒
1352078
計算要した時間は 1.78009104729 秒
```

(3) を使って計算した場合は計算時間は0.0002秒なのに対して，(4) を使った計算では1.78秒かかっています。これは漸化式 (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，漸化式 (4) から C(5,2) を計算するのに，次のような膨大な計算を行っていることになります：

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

こんなやり方では，もっと大きな組み合わせの数 C(54,24)=1402659561581460 を計算するには約 100 年もかかってしまいます。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，事実上計算不可能な定義になってしまうことがあるので注意が必要です。

20.9 練習問題

与えられた自然数 n に対して

$$\sum_{j=1}^{n} \frac{1}{j^2} \tag{5}$$

を返す関数を再帰的に定義したい。上の和を返すような関数 Basel_sum(n) を定義しなさい。

- ファイル名：**Basel_sum.py**

```
1  def Basel_sum(n):
```

```
2     if n<2:
3         return 1
4     else:
5         ****** ************************
6
7 print Basel_sum(10)
8 print Basel_sum(100)
```

21　知っておくと便利なこと ——— 文字列から式や文を作り出す

以前説明したように，Sage や Python では式 (expression)，文 (statement) があり，これらは明確に区別されています。たとえば，ans = 2^2^2 としたとき ans は変数名ですが，クォーテーションで囲んだ'ans' は文字列です。また a=1 は文で'a=1' は文字列。1+1 は式ですが'1+1' は文字列です。

ここで一つ疑問が生じます，文字列を式や文に変換するにはどうしたらいいでしょう？そのようなときに使うのが eval 関数と exec 関数です。

eval 関数と exec 関数

- eval 関数は文字列で表されている式 (expression) を評価してその値を返す。
- exec 関数は文字列で表されている文 (statement) を実行する。

例：

```
>>> eval('1+3')          #1+3 が評価されてその値が返される
4
>>> exec('a=3')           #a=3 が実行される
>>> print a
3
```

ここで 1+3 は式，a=3 は文であり，'1+3' と'a=3' は文字列であることに注意。

eval を使うと文字列を変数名に変えることができます。例えば次のようなプログラムを作ったとしましょう：

```
def func1():
    return 2**2**2

def func2():
    return 'hogehoge'

def func3():
    return type(1.4)

def func4():
    return [1,2,3,4]

print func1()
print func2()
print func3()
print func4()
```

実行結果の例

```
16
hogehoge
<type 'float'>
[1,2,3,4]
```

func1，func2，func3，func4 と 4 つの関数を作ったのでこれをプリントする為に 4 回関数を呼び出す必要がありました。これを自動化するために，関数名からなる文字列を生成して eval で評価します：

```
def func1():
    return 2**2**2
```

```
3
4  def func2():
5      return 'hogehoge'
6
7  def func3():
8      return type(1.4)
9
10 def func4():
11     return [1,2,3,4]
12
13 for i in range(1,5):
14     print eval('func'+str(i)+'()')
```

実行結果は先ほどと同じになります。ここで str 関数は中身を文字列に変換する関数だったことを思い出しましょう。

22　Set 型

22.1　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。ですのでこれは数学の集合と類似の概念です。集合は set([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：

```
1  >>> a = set([1,3,4])      # 集合aを定義
2  >>> b = set([3,7])        # 集合bを定義
3  >>> c = a | b             # cを集合aとbの和集合とする
4  >>> c
5  set([1, 3, 4, 7])
6  >>> d = a & b             # dを集合aとbの共通部分とする
7  >>> d
8  set([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：

```
1  >>> a.union(b)            #a.union(b)は
2  set([1, 3, 4, 7])         #aとbの和集合を返す
3  >>> a.intersection(b)     #a.intersection(b)は
4  set([3])                  #aとbの共通部分を返す
5  >>> a
6  set([1, 3, 4])            # 上の操作でaの値が変わるわけではない
7  >>> b
8  set([3, 7])               #bもかわらない
9  >>> a |= b                #aにbの要素をすべて追加する
10 >>> a                     #aには
11 set([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
set([8, 1, 3, 4])     # a に要素 8 が追加された
>>> a.remove(1)       # a から要素 1 を除く
>>> a
set([8, 3, 4])        # a から 1 が削除された
```

22.2 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 Python らしく 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。）

22.2.1 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 set([a+b,a-b,b-a,a*b,1.0*a/b, 1.0*b/a]) # 四則演算からできる集合
    elif b == 0:            # b=0 のときは
        return set([a, -a, 0])
    else:                   # a=0 のときは
        return set([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

```
set([0.75, 1, 7, 12, 1.3333333333333333, -1])
```

上のプログラムで 1.0*あるのは演算『/』を浮動小数点数の割り算にしたいからです。

22.2.2 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。
- $(a \circledast c) \ominus d$ ： a,c を先に計算してから次に d 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 set([a+b,a-b,b-a,a*b, 1.0*a/b,1.0*b/a])
    elif b == 0:
        return set([a,-a,0])
    else:
        return set([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

```
set([-1.25000000000000, 3.25000000000000, 1.25000000000000, 1,
8.00000000000000, 9, 10.0000000000000, -3.55555555555556, 144, 17, 20,
25, -0.800000000000000, 0.562500000000000, 0, 32, 0.888888888888889, 40,
7, -1.75000000000000, 52, 1.75000000000000, 1.12500000000000,
4.44444444444444, 72, 0.307692307692308, -7, -32, 0.800000000000000,
3.55555555555556, -20, 0.111111111111111, -9, -8.00000000000000,
1.77777777777778, -1, 6.25000000000000])

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**

```
# -*- coding:utf-8 -*-
def nikou(a,b):
    if a !=0 and b!=0:
        return set([a+b,a-b,b-a,a*b, 1.0*a/b,1.0*b/a])
    elif b == 0:
        return set([a,-a,0])
    else:
        return set([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
```

```
14      for i in nikou(b,c):
15          ResultSet |= nikou(i,a)
16      for i in nikou(a,c):
17          ResultSet |= nikou(i,b)
18      return ResultSet    # ResultSetを返す
19
20  number = 0      # numberという変数を作り0にセットする
21
22  for i in range(10):
23      for j in range(i,10):
24          for k in range(j,10):
25              if 10 in sankou(i,j,k):    # もし10がsankou(i,j,k)の要素なら
26                  print i,j,k      #i,j,kを表示して
27                  number = number+1    #numberを1増やす
28
29  print '10を作れる数字の三つ組みの数は', number, '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
```

22.2.3 4 つの数字の四則演算

つぎに 4 つの数 a,b,c,d から作れる数の集合を返す関数を作りたい。そのような演算には次の 2 通りある。

(A) 3 つの数の演算を行った後に，残りの数との演算を行う場合
(B) 2 つの数の計算をそれぞれ行って，次にそれらの演算を行う場合

例えば

(a+d*c)*b　　(a*d)+(b/c)

の前者は (A) の場合で，後者が (B) の場合である。

(A) の手順で作られる数字の集合を返す関数を yonkouA(a,b,c,d) とすると，これは次のように定義すればよい

```
1   def yonkouA(a,b,c,d):
2       ResultSet = set([])
3       for i in sankou(a,b,c):      # a,b,cから作られる数iに対して
4           ResultSet |= nikou(i,d)  # iとdの二項演算から作られる数をResultSetに追加
5       for i in sankou(a,b,d):
6           ResultSet |= nikou(i,c)
7       for i in sankou(a,c,d):
8           ResultSet |= nikou(i,b)
9       for i in sankou(b,c,d):
10          ResultSet |= nikou(i,a)
11      return ResultSet
```

(A) の演算を用いて 10 を作ることが出来る 4 つの数をすべて列挙するプログラムは次のようになる。

- ファイル名：**yonkou1.py**

```
1   # -*- coding:utf-8 -*-
```

```
2  def nikou(a,b):
3      if a !=0 and b!=0:
4          return set([a+b,a-b,b-a,a*b, 1.0*a/b,1.0*b/a])
5      elif b == 0:
6          return set([a,-a,0])
7      else:
8          return set([b, -b, 0])
9
10 def sankou(a,b,c):        # a,b,cの四則演算で作られる集合を返す
11     ResultSet = set([])   # ResultSetを空のリストとする
12     for i in nikou(a,b):         # a,bから作られる数iに対して
13         ResultSet |= nikou(i,c) # i,cの演算から作られる集合をResultSetに追加
14     for i in nikou(b,c):
15         ResultSet |= nikou(i,a)
16     for i in nikou(a,c):
17         ResultSet |= nikou(i,b)
18     return ResultSet    # ResultSetを返す
19
20 def yonkouA(a,b,c,d):
21     ResultSet = set([])
22     for i in sankou(a,b,c):       # a,b,cから作られる数iに対して
23         ResultSet |= nikou(i,d)  # iとdの演算から作られる数をResultSetに追加
24     for i in sankou(a,b,d):
25         ResultSet |= nikou(i,c)
26     for i in sankou(a,c,d):
27         ResultSet |= nikou(i,b)
28     for i in sankou(b,c,d):
29         ResultSet |= nikou(i,a)
30     return ResultSet
31
32 number = 0
33
34 for i in range(10):
35     for j in range(i,10):
36         for k in range(j,10):
37             for l in range(k,10):
38                 if 10 in yonkouA(i,j,k,l):
39                     print i,j,k,l
40                     number = number+1
41
42 print '10を作れる数字の三つ組みの数は', number, '個'
```

実行結果の例

```
0 0 1 9
0 0 2 5
...
...
8 9 9 9
9 9 9 9
10 を作れる数字の 4 つ組みの数は 545 個
```

さて，これで (A) の演算で 10 を作ることが出来る数字の 4 つ組みを確定することが出来た。しかし上の実行結果には手順 (B) で 10 を作ることができる数字が含まれていない。例えば 1114 は (1+1)*(1+4)=10 のように 10 を作ることができるが上の実行結果にはない。

そこで次に，計算 (B) の計算で作られる数の集合を返す関数 yonkouB(a,b,c,d) を定義し，4 つの数字から四則演算で作られるリストをすべて列挙し，その個数を求めたい。yonkouB(a,b,c,d) は次のように定義すればよい：

```
def yonkouB(a,b,c,d):
    ResultSet = set([])            # ResultSetを空のリストとする
    for i in nikou(a,b):           # aとbから作られる数をiとする
        for j in nikou(c,d):       # cとdから作られる数をjとする
            ResultSet |= nikou(i,j) # iとjから作られる数の集合をResultSetに加える
    for i in nikou(a,c):
        for j in nikou(b,d):
            ResultSet |= nikou(i,j)
    for i in nikou(a,d):
        for j in nikou(b,c):
            ResultSet |= nikou(i,j)
    return ResultSet                # ResultSetを返す
```

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すればよい：

- ファイル名：**make10.py**

```
# -*- coding:utf-8 -*-
def nikou(a,b):
    if a !=0 and b!=0:
        return set([a+b,a-b,b-a,a*b, 1.0*a/b,1.0*b/a])
    elif b == 0:
        return set([a,-a,0])
    else:
        return set([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を空のリストとする
    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

def yonkouA(a,b,c,d):
    ResultSet = set([])
    for i in sankou(a,b,c):     # a,b,cから作られる数iに対して
        ResultSet |= nikou(i,d) # iとdの二項演算から作られる数をResultSetに追加
    for i in sankou(a,b,d):
        ResultSet |= nikou(i,c)
    for i in sankou(a,c,d):
        ResultSet |= nikou(i,b)
    for i in sankou(b,c,d):
        ResultSet |= nikou(i,a)
    return ResultSet

```

```
32 def yonkouB(a,b,c,d):
33     ResultSet = set([])            # ResultSetを空のリストとする
34     for i in nikou(a,b):           # aとbから作られる数をiとする
35         for j in nikou(c,d):       # cとdから作られる数をjとする
36             ResultSet |= nikou(i,j) # iとjから作られる数の集合をResultSetに加える
37     for i in nikou(a,c):
38         for j in nikou(b,d):
39             ResultSet |= nikou(i,j)
40     for i in nikou(a,d):
41         for j in nikou(b,c):
42             ResultSet |= nikou(i,j)
43     return ResultSet                 # ResultSetを返す
44
45 def yonkou(a,b,c,d):
46     ResultSet = set([])
47     ResultSet |= yonkouA(a,b,c,d)
48     ResultSet |= yonkouB(a,b,c,d)
49     return ResultSet
50
51 counter = 0
52
53 for i in range(10):
54     for j in range(i,10):
55         for k in range(j,10):
56             for l in range(k,10):
57                 if 10 in yonkou(i,j,k,l):
58                     print i,j,k,l
59                     counter = counter+1
60
61 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。それは，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ためです。ただ，10 に非常に近い数は 10 であると認識されます：

```
1 >>> 0.9999999999999999 == 1
2 True
3 >>> 0.999999999999999 == 1
4 False
```

22.3 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 回ずつ使った四則演算（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。

第 II 部
数式処理システム SageMath

23　数式処理システム SageMath とは

SageMath(以下，Sage と略す) 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 言語で書かれていて，Sage のプログラムは 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。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。Sage は Linux, Mac, Windows で利用可能です。

24　Sage 実行方法

Sage の実行方法は主に次の 4 種類あります：

1. インタラクティブ・シェル (端末からコマンド `sage` で起動する CUI)
2. Sage のプログラムが書いてあるファイルを呼び出して実行 (端末で `sage ファイル名.sage` とする)
3. Sage ノートブックを使う (インタラクティブ・シェルから `notebook()` で起動)
4. Sage のインタラクティブシェルからロードする (`sage:load(‘‘ファイル名.sage’’)`)

以下では，まず，1 と 2 について解説します。

24.1　Sage のインタラクティブシェルの起動

Sage がインストールされている PC では，端末（ターミナル）から sage と入力することによって Sage を起動することができます。端末で

```
user@debian:~$ sage              # SageMathを起動$
```

と入力すると

```
┌────────────────────────────────────────────────────────────────────┐
│ SageMath version 8.2, Release Date: 2018-05-05                     │
│ Type "notebook()" for the browser-based notebook interface.        │
│ Type "help()" for help.                                            │
└────────────────────────────────────────────────────────────────────┘
sage:
```

と表示され，Sage のインタラクティブ・シェルが起動します。ここでは Python のシェルと同じ事ができますが，それに加えて Sage ならではの機能が使えます。試しに 1+2 や factor(2010) 等と入力してみましょう：

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

ここで factor というのは，因数分解を行う関数で，Sage にははじめから組み込まれています。

Sage を終了して元の端末の状態に戻るには『exit』または『quit』を入力します。

```
sage: exit
Exiting Sage (CPU time 0m0.06s, Wall time 2m8.71s).
```

24.2 Sage のプログラムを作成して実行する

Python と同様に，Sage のプログラムをファイルに書いて端末から実行することができます。具体的な手順は次の通りです。

1. Sage プログラムをテキストエディタで書いて保存。ファイル名は『ファイル名.sage』とします。
2. 端末を起動し，ファイルの保存されているディレクトリへ移動し，

   sage ファイル名.sage

   と入力する。[*8]

上の手順を試してみましょう。まず次のファイルを作ります。

- ファイル名：**testfactor.sage**

```
a = factor(2014)          # 2014の因数分解の結果をaとする
print a                   # aをプリントする。
```

そして，端末でファイルの位置までディレクトリを移動して次のように実行します：

```
$ sage testfactor.sage
2 * 19 * 53
$
```

この場合，Python の場合と同様に，出力したい値は print で書かなければ表示されません。

Python と同じ事ですが，出力結果をファイルに保存したい場合，次のようにします：

```
$ sage testfactor.sage > testfactor.txt
$
```

すると実行したディレクトリに testfactor.txt というファイルが作られて，実行結果がこのファイルに記録されます。

25 Sage の Web ページについて

Sage に関する情報は公式ウェブサイト

http://www.sagemath.org/

から得ることができます。トップページのメニューから次の情報にアクセスできます：

- Try Sage Online：インターネットから Sage 利用することができます。
- 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。

[*8] ⊛ ファイルと異なるディレクトリから実行する場合はディレクトリも付けたファイル名を書きます。

- Download：Sage 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)。
- Sage Feature Tour：いろいろな活用方法が紹介されています。

26　他の環境での Sage の利用

26.1　SageMathCloud

インターネットに接続されている環境では Sage の Online 版が使えます。Sage の Web サイト `http://www.sagemath.org/`へ行き，Try Sage Online をクリックします。自分用のアカウントを作成してサインインすれば，Sage ノートブックが利用できるようになります。

26.2　Windows に Sage をインストールする

SageMath8.0 から Windows がネイティブにサポートされました。インストールは次のように行います。

1. SageMath のダウンロード用のミラーサイト

   `http://ftp.riken.jp/sagemath/win/index.html`

   から `SageMath-**.exe` をダウンロードします (xx はバージョンによって異なります。)
2. ダウンロードが完了したらファイルを実行することにより，インストールが始まるので画面の指示に従ってインストールします。

26.3　Mac での利用

以下は `http://www.sagemath.org/mirror/osx/README.txt` の解説を翻訳したものです[*9]。
自宅の PC が Max OS X の場合，インストールは次のように行います。

1. まず Sage の Web サイトの download から適当なサーバーを選んで sage の dmg ファイルをダウンロードします。新しい mac なら CPU は intel だと思いますので，**intel** をクリック，OS のバージョンが 10.10 で 64bit マシンなら `sage-xx-x86_64-Darwin-OSX-10.10_x86_64-app.dmg` をダウンロードします。ファイル名に`-app` がついているのは通常の Mac のアプリケーションとして起動するファイルで，`-app` がついていないものは，伝統的な Unix command-line として起動するファイルだそうです。おそらく通常は名前に`-app` がついているものをダウンロードすればよいとおもいます。
2. dmg ファイルをダブルクリックします。
3. sage フォルダーを適当にアプリケーションのフォルダにドラッグします。
4. finder でコピーした sage フォルダーへ移動して『sage』のアイコンをダブルクリック。
5. Select to run it with "Terminal":

   Choose Applications, then select "All Applications" in the "Enable:" drop down. Change the "Applications" drop down to "Utilities". On the left, scroll and select "Terminal". Click "Open", then in the next dialog select "Update".
6. Sage の Window がポップアップします。
7. Sage ノートブックを起動する場合は，`notebook()` とタイプし，firefox か safari で URL：`http://localhost:8000` を入力します。

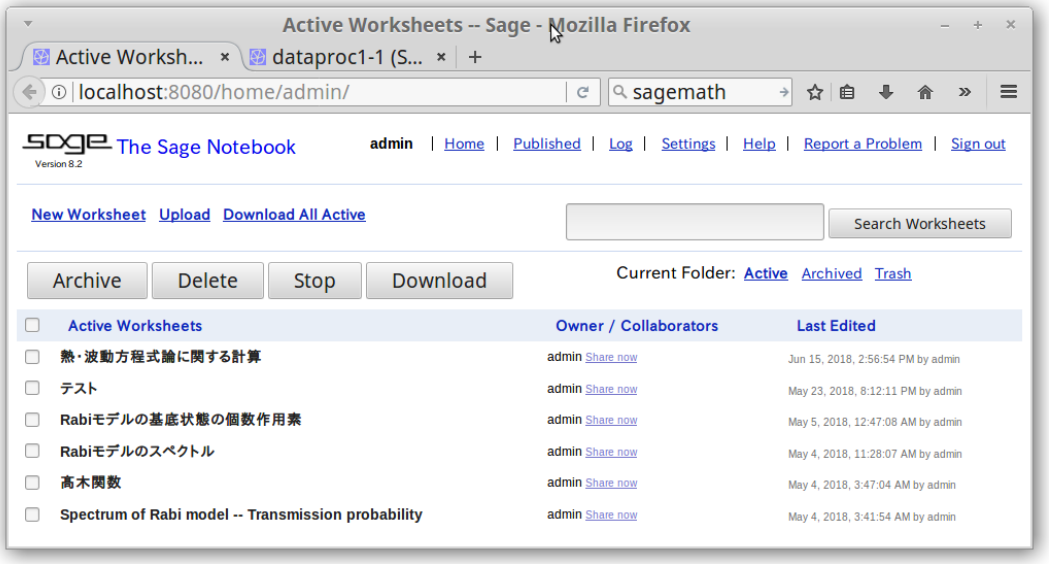
[*9] 私自身 Mac を持っていないので動作確認していません。うまくいかないことがあるかもしれませんが，各自がんばってください。

27　Sage ノートブック

端末から Sage を起動した後に Sage のコマンドラインから `notebook()` を入力することで Sage ノートブックが起動します。

```
1 sage: notebook()
```

すると firefox などのウェブブラウザが起動し次のような画面が現れます。これが Sage のノートブックです。初回起動時にパスワードが聞かれますが，適当に覚えやすいものを入力しておきます[*10]。



この画面で worksheet が開いていなければ，New Worksheet をクリックして開きます。最初にワークシートの名前を決めます。Sage のプログラムは四角い枠 (セル) の中に書きます。試しに $1+2$ を計算してみましょう。セルに `1+2` と書き，次のどちらかの方法でセルに書かれたプログラムを実行します。

- `shift+enter`
- セルの左下にある `evaluate` をクリック

Sage ノートブックを終了するには，ブラウザを閉じ `notebook()` と入力した端末で `ctrl+c` を押すと Sage のインタラクティブシェルに戻ります。`exit` と入力すれば Sage が終了します。

28　Sage での数の取り扱い

28.1　四則演算

Sage の四則演算は，基本的には Python と同じすが，記号『`/`』は異なります。整数 a, b に対して Python2 では `a/b` は商を返しましたが，Sage では a/b が割り切れないときには，有理数 a/b となります。また，冪の記号は Python では a^b は `a**b` でしたが，Sage では『`a^b`』のように書きます。

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

[*10] 毎年，授業でユーザーネームとパスワードを忘れる人がいるので，ここでは共通になるように，`username = admin, password = sagesage` としてください。

28.2 数の表示

Sage では自然数，有理数，有限体を取り扱うことができ，無理数は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：

```
pi
```

実行結果の例

```
pi
```

数値を表示するには n() という命令を使います：

```
n(pi)
```

実行結果の例

```
3.14159265358979
```

円周率を 30 桁だけ表示するには：

```
n(pi, digits=30)
```

実行結果の例

```
3.14159265358979323846264338327950288 4197
```

自然対数の底についても同様に

```
n(e)
```

実行結果の例

```
2.71828182845905
```

平方根 (square root) は次のようになります：

```
sqrt(3)
```

実行結果の例

```
sqrt(3)
```

平方根の計算は厳密に行われ

```
sqrt(3)*sqrt(3)
```

実行結果の例

```
3
```

となります。e や pi と同様に $\sqrt{3}$ の近似値は n(sqrt(3)) で計算します。

28.3 Sage で文字 n を使うときの注意

Sage では文字 n に，n(a) で『a の数値を返す』という命令があらかじめ割り当てられているので，文字 n は特に理由がなければ使わないようにしましょう。

28.4 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: exp(pi*I)
-1                    # Eulerの公式が使える
```

28.5　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。

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	$\prod_{p\geq 3;\ 素数}\frac{p(p-2)}{(p-1)^2}$

28.6　数の精度

数値計算をするときには常に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 = var('x')  # xを形式的な文字と定義する。
sage: x in SR       # 形式的に取り扱える文字
True
```

円周率の精度は次のようになります：

```
print pi
print RDF(pi)
print RR(pi)
print RealField(200)(pi)
```

実行結果の例

```
pi
3.14159265359
3.14159265358979
3.14159265358979323846264338327950288419716939937510582097494
```

誤差が生じるのは次のような場合です，

```
sage: a = RR(pi); exp(a*I)
-1.00000000000000 + 1.22464679914735e-16*I
```

厳密には $e^{i\pi} = -1$ なので実数となるはずなのに，上の実行結果では虚数部分 `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，この絶対値は $10^{-16}i$ 程度なので非常に小さいですが，答えが実数でないために，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`) を使用とした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エラーとなります。複素数値を含む関数の数値計算を取り扱う場合は，そのような注意を常にしておかなければなりません。

29　カーネルの初期化

Sage notebook では `a=3` と入力すると，`a` が 3 であるという情報を持ち続けています。次に `a=5` と入力すれば，`a` の値は 5 に変わります。Sage が記憶している 変数の情報をリセット するにはノートブックから『Action...』→『Restart worksheet』をクリックします。

30　Sage ノートブックの補完機能

端末では，ディレクトリ名やファイル名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，Sage ノートブックも同様の機能を持っています。Sage ノートブックで `dif` と入力して `Tab` キーを押してみましょう。`diff` と `differences` の 2 つの候補が出てきます。`diffe` まで入力してから `Tab` キーを押すと，`differences` が自動的に出てきます。ここで `diff` は微分をする命令で，`differences` は数列の差をとる命令です。

31　ノートブックの出力の整形

ノートブック画面の `Typeset` にチェックマークを入れると，命令の実行結果が**数学の書式で整形**された形で出力されます。`Typeset` にチェックマークを入れてからセルで次を実行してみましょう：

```
  integral(sin(x)*e^x,x)
```

`Typeset` にチェックマークを入れない場合はどのように表示されるだろうか？（試してみてください）

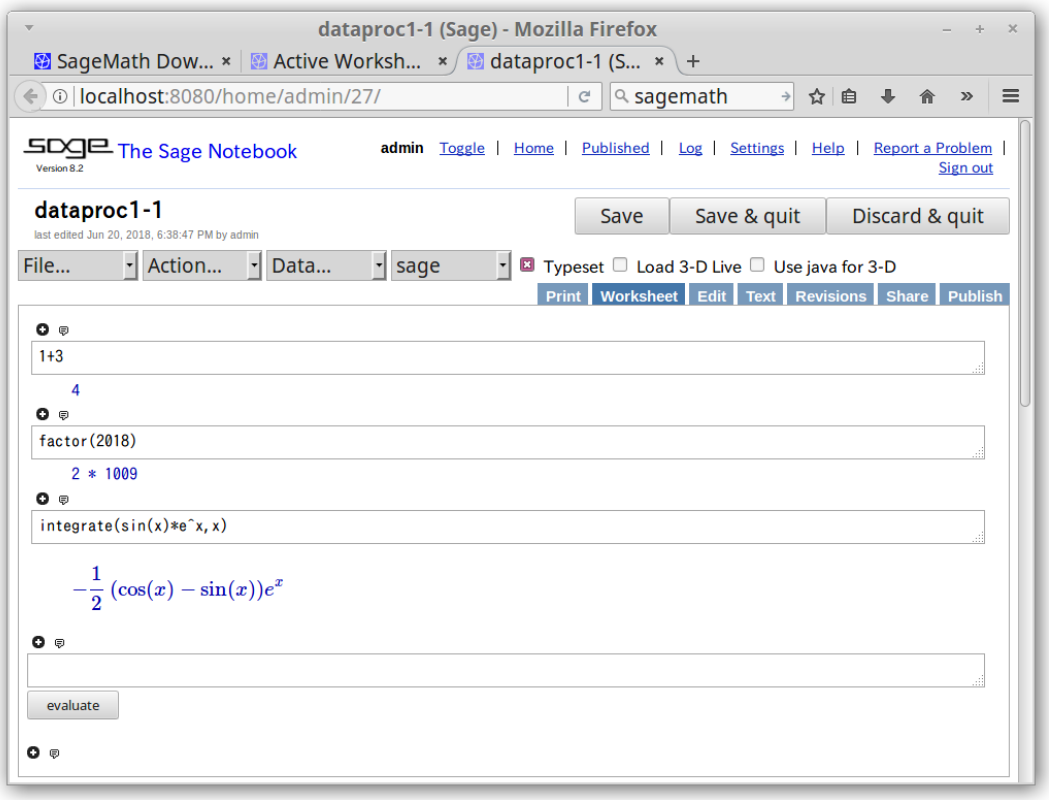


図 8　出力結果：Typeset

32　Sage ワークシートの保存

Sage ノートブックのワークシートはファイルに保存することができます。ワークシートは Sage worksheet 形式（sws）で保存されます。ワークシートの保存は Web ブラウザ[*11]でファイルをダウンロードと同じ要領で行います[*12]。Sage worksheet を保存する手順は次の通りです[*13]。

1. Sage ワークシートを開いている状態で，[File...] → [Save worksheet to a file...] を順にクリック。
2. 保存するファイル名を聞かれるので，入力して OK ボタンを押す。
3. Web ブラウザで通常のファイルのダウンロードの画面になるので，『ファイルを保存する』を指定して OK ボタンを押す。
4. 保存するフォルダを指定する。

今回はファイル名を `sage01.sws` として，『他のフォルダ』のデスクトップを選び，デスクトップにあるディレクトリ `dataproc1` に保存しておきましょう。ファイル名の拡張子は `sws` です。拡張子を変えてはいけません。

次に，ファイルに保存した worksheet のファイルを読む込む手順は次の通りです。

1. Sage notebook のトップページの左上にある upload をクリックする。
2. Browse your computer to select a file to upload: の下の「参照」を押して，読み込むファイルを選択

[*11] いまは firefox
[*12] (ファイル→名前を付けてページを保存) ではありません！
[*13] その前にブラウザの設定を編集し，ダウンロードの設定で『ファイルごとに保存先を指定する』となるようにしておきましょう。

3. ファイルを選んだら [upload] ボタンを押す。

33　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：

```
sum?
```

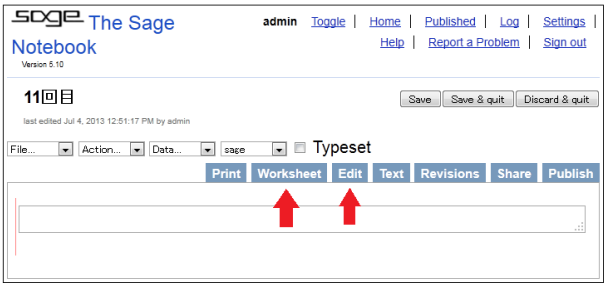
和 sum の使い方とその例が表示されたはずです。さらに関数 sum を定義しているソースコードを表示するには次のようにします：

```
sum??
```

34　Sage ワークシートの活用

34.1　ワークシートのソースとその編集

Sage ノートブックのセルの欄外には Word のようにタイトルや説明などを書くことができます。ここではその方法を紹介しましょう。まず Sage ノートブックを開き New Worksheet を作ります。タブに『Worksheet』と『Edit』という項目がありますが，これらクリックすることでワークシートとそのソースを切り替えます。



それでは Edit を押してみましょう。すると次のようなものが表示されます。

```

{{{id=1|

///
}}}
```

これがワークシートのソースです。Worksheet のタブをクリックすると先ほどのワークシートに戻ります。さて，試しに Worksheet のセルに

```
integrate(sin(x),x)
```

と入力して実行してから，Edit タブをクリックしてソースを表示してみましょう。するとソースは次のように変わっています：

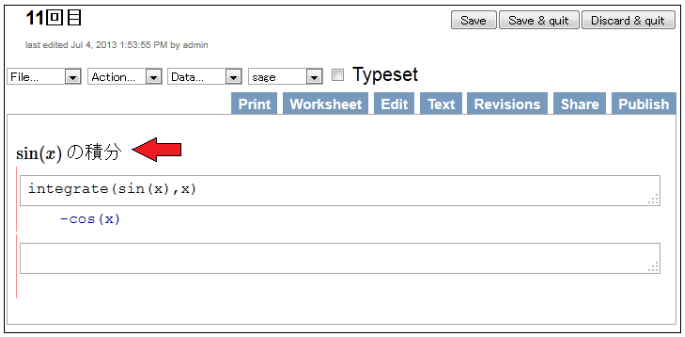
```
                                # ここは欄外
{{{id=1|                        # ここから一つ目の項目がはじまる
integrate(sin(x),x)             # セルの内容
```

```
///                              # 入力と出力の区切り
-cos(x)                          # セルの実行結果
}}}                              # 項目終了
                                 # ここは欄外
{{{id=2|          # 2つ目の項目の始まり
                  # 二つ目のセルの内容
///               # 区切り
}}}               # 2つめの項目の終わり
```

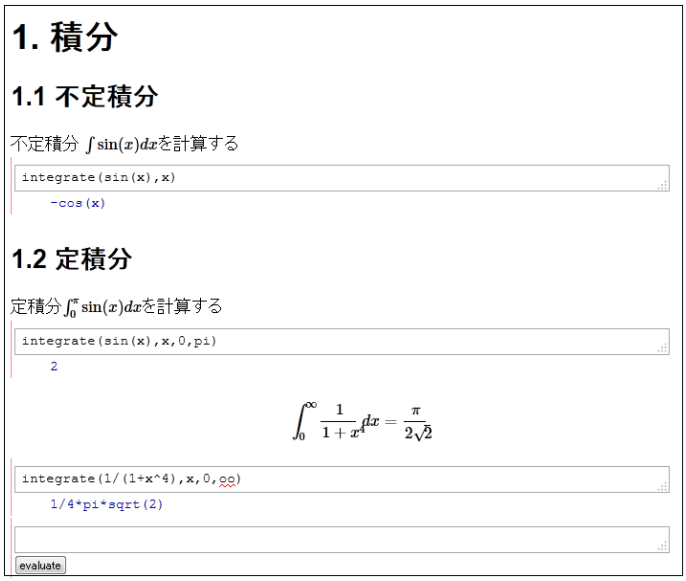
{{{id=1|の直後に，一つ目のセルの内容と実行結果が追加されているのがわかります。また，二つ目のセルが自動的に作られたのでその文だけソースが増えています。さて，上のテキストの 1 行目に

```
$\sin(x)$ の積分
```

と書いて，『Save changes』のボタンをクリックすると内容が更新されてワークシートが表示されます。下の図の矢印の部分のようになっていたら成功です：



このようにして，セルの欄外にタイトルを書いたり，解説や補足を書くことができます。また，簡単な LaTeX の形式で数式を書くこともできます。数式を書くときは LaTeX と同じようにドルマーク $ $で囲むか \[\] の中に書きます。\begin{equation}...\end{equation} 環境を使うこともできます。セルの欄外には html 形式を使うことができます。例えば html のコマンドを使って見出しを付けることで見やすいワークシートを作ることができます：



上のワークシートのソースは次のようになっています：

```
<h1>1. 積分</h1>
```

```
<h2>1.1 不定積分 </h2>
不定積分 $\int \sin(x)dx$ を計算する

{{{id=1|
integrate(sin(x),x)
///
-cos(x)
}}}

<h2> 1.2 定積分 </h2>
定積分$\int_0^\pi \sin(x) dx $ を計算する

{{{id=2|
integrate(sin(x),x,0,pi)
///
2
}}}

\begin{equation}
 \int_0^\infty \frac{1}{1+x^4}dx = \frac{\pi}{2\sqrt{2}}
\end{equation}

{{{id=3|
integrate(1/(1+x^4),x,0,oo)
///
1/4*pi*sqrt(2)
}}}

{{{id=4|

///
}}}
```

上のソースでは，<h1>見出し 1</h1>で 1 番目の見出しを書いています，2 番目の見出しは<h2> ... </h2>で囲んで書きます。Worksheet では html 形式のタグはすべて使えるのでフォントサイズを変えたり，太字にしたり，色を付けたりすることもできます。html の書き方はインターネットにいくらでもあるので必要に応じて調べてみましょう。

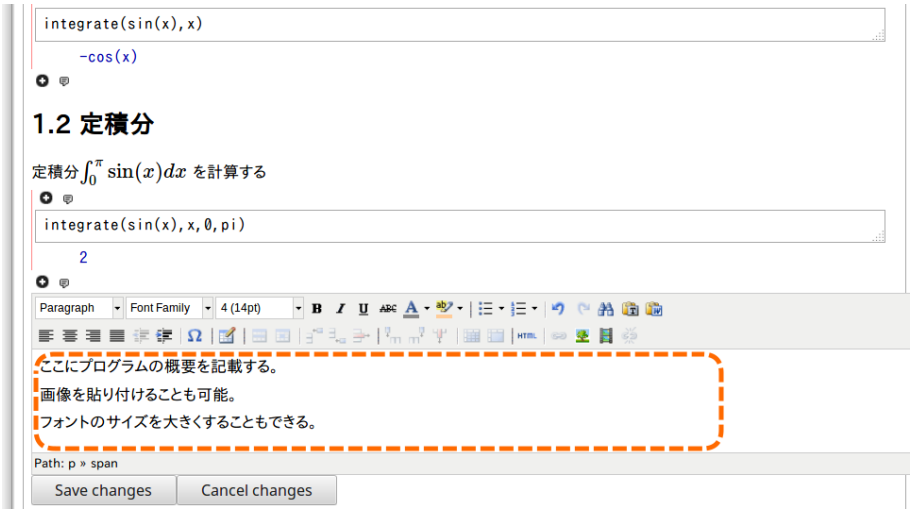
34.2 Sage ワークシートのドキュメントとしての機能

上のように，Sage ワークシートのソースを直接編集して，欄外にドキュメントを書くことができますが，もっと便利な方法があります。右の図のように，セルの外側にマウスカーソルを置くと青紫色の線が現れます：このとき，

Shift ＋ マウス左クリック

を押すと，次の図のようにテキスト編集画面が現れます。

ここにテキストを書いてフォントの種類やサイズを変えたり，番号付けしたり，LaTeX 形式で数式を書いたりすることができます。



画像ファイルを貼り付けることもできます。記入したら，『Save Changes』をクリックすれば編集が反映されます。また，すでに編集した欄外の項目であれば，ダブルクリックするだけでテキスト編集画面が現れます。

このように，Sage Worksheet は，計算できる上に，ドキュメント形式としても十分な機能を持ってるので，計算可能なドキュメント であるといえます。

35　練習問題

Sage のワークシート作成，ファイルに保存してみましょう。Sage ワークシートのファイル名は `sagews01.sws` とすること。ファイルを保存したら Sage ノートブックの画面の Upload からアップして，ワークシートが正常に読み込まれているか確認してみましょう。

36　2 つの関数

ここでは，以前に Python で定義した関数とは異なる『関数』について解説します。まず，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)             # 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文字式としての取り扱い方を学習するからだとおもいます。たとえば

$$4x+3, \quad x^2, \quad \sin(x), \quad \frac{1}{x^2+1} \tag{6}$$

はどれも文字式による関数です。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。そして，これらは x が何者であるかはとりあえずは特定せずに x の文字式としての扱われます。

Sage では，文字式としての関数に対応するものを定義することができます。

文字式としての関数の定義 1

```
1  var('x')          #  xを変数＝不定元にする宣言
2  f = x^2           #  文字式x^2を変数fに代入
```

上のプログラムの 1 行目は文字 `x` を変数として取り扱いますという宣言です。これにより `x` の文字式が扱えるようになります。2 行目で関数 $f=x^2$ を定義しています。

上に続いて次を実行します

```
1  print f             #  fをプリント
2  print f(x=5)        #  x=5のときのfをプリント
3  print type(f)       #  fの型を調べる
4  print type(x)
```

実行結果の例

```
x^2
25
<type 'sage.symbolic.expression.Expression'>
<type 'sage.symbolic.expression.Expression'>
```

このようにして定義した関数 f の特定の x に対する値を計算するときには 2 行目のように書きます。3,4 行目の結果は，f と x のデータの型は sage.symbolic.expression.Expression というものになっています。一方，上で定義した fib のデータの型は'function' です。

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

```
1 var('x')
2 f(x) = x^2          # 変数の指定(x)があるのが前と異なる
3
4 print f
5 print f(5)          # これで関数の値が返される
6 print type(f)
```

実行結果の例

```
x |--> x^2
25
<type 'sage.symbolic.expression.Expression'>
```

また，2 変数 x,y の関数を定義するには次のようにします

```
1 var('x y')      #x, yを文字式として取り扱う宣言
2 g = (x+y)^2      # 関数gの定義
3 print g
4 print g(x=3, y=6)      #x=3, y=6のときのgの値
5 print g(y=3)      #y=3のときのgの値
```

実行結果の例

```
(x + y)^2
81
(x + 3)^2
```

36.1 Sage で使える関数

Sage では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三角関数とその逆関数 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，tanh 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。

37 グラフの描画とデータの可視化

データの可視化を解説します。Sage では手軽に高機能なグラフ描画機能を利用することが出来ます。以下では出力は Sage ノートブック上で実行していると仮定しています。

37.1 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), (x,a,b))
```

- 関数 $f(x)$ を x の範囲 $[a,b]$ で描画する。

例えば $\sin(x)$ を区間 $[-6,6]$ でプロットするには次のようにします：

例 1：
```
plot(sin(x), (x,-6,6))
```

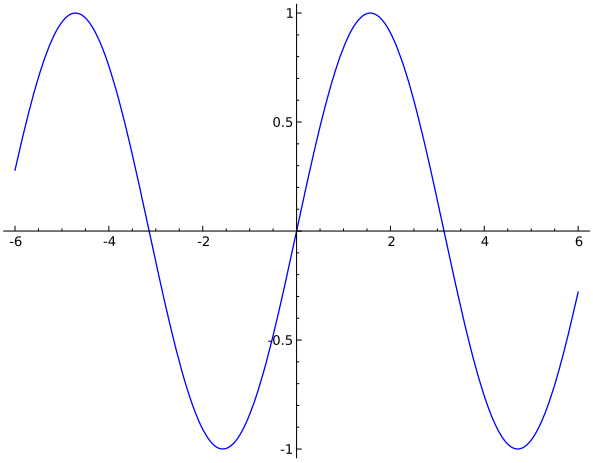


図 9　出力結果：$\sin(x)$ のグラフ

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**：
```
plot(1/x, (x,-2,2))
```

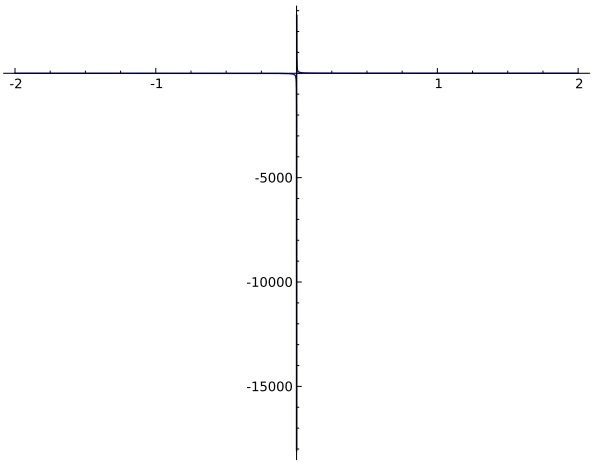


図 10　出力結果：$1/x$ のグラフ

グラフの縦軸が異常に大きい数値になって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 $[c,d]$ の範囲で描画

```
1 plot(f(x), (x,a,b), ymin=c, ymax=d)
```

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

関数 $1/x$ のグラフを値域を $[-3,4]$ に制限して描画します。

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

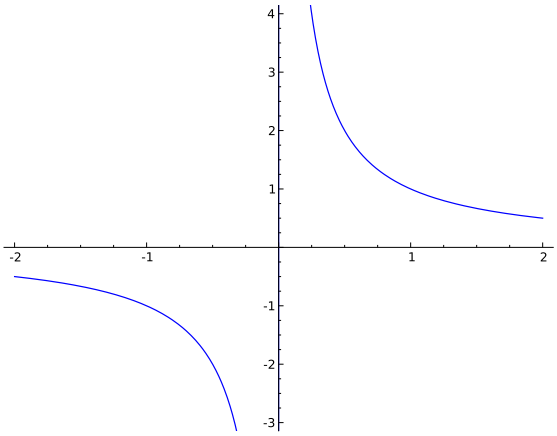


図 11　出力結果：$1/x$ を値域 $[-3,4]$ に制限したグラフ

複数のグラフを一度に表示するには次のようにします：

関数 $f(x), g(x), h(x)$ のグラフを重ねて描画

```
1 plot( (f(x),g(x),h(x)), (x,a,b) )
```

三つ以上でも同様にカンマで区切って描くことができます。例えば 3 つの関数 $\sin(x), \cos(x), \tan(x)$ を区間 $[-7,7]$，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)), (x,-7,7), ymin=-5,ymax=5)
```

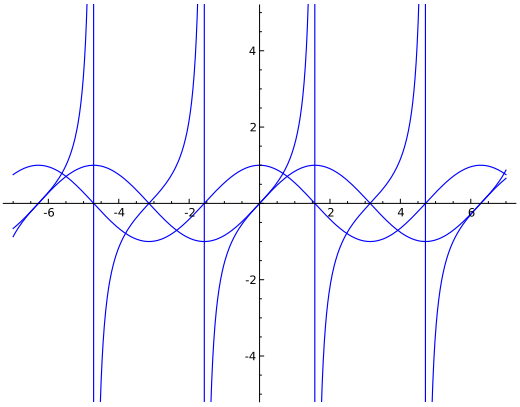


図 12　出力結果：$\sin(x), \cos(x), \tan(x)$ のグラフ

plot——様々なオプション

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。そのときの書式は次の通りです

```
plot( f(x), (x,a,b), color='グラフの色',
      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 で軸のラベルを指定。
- 座標軸を消すには axes=False を追加します。
- 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：
```
plot(sin(x^2),(x,0,5), color='red', axes_labels=['time','amplitude'],
     legend_label='$\sin(x^2)$')
```

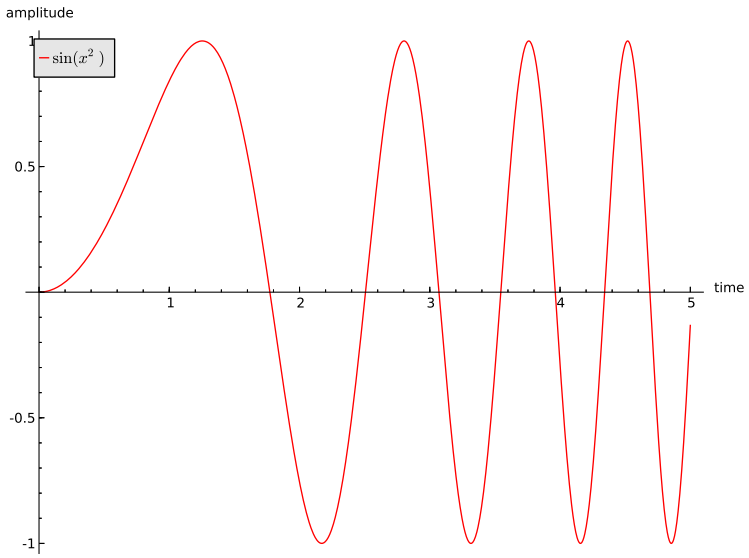


図 13　出力結果：$\sin(x^2)$ のオプション付きプロット

37.2　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：

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

実行結果の例

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

37.3　グラフ描画：応用編

まず show() の使い方を説明します。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が show() です。以下を実行してみましょう：

show の使い方 1

```
1 p1 = plot( sin(x), (x,-6,6) )
2 show(p1)
```

- 1 行目で $\sin(x)$ のグラフデータを変数 p1 に代入しています。
- 2 行目で p1 を画面に表示させています。
- 2 行目は p1.show() としても同じです（試してみましょう）。

次のように show() の中にオプションを書くこともできます：

show の使い方 2

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

次の例のように，それぞれのグラフに色を付けて重ねて表示することができます：

例 6：

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

上の命令では $\sin, \cos, \tan$ のそれぞれのグラフは個別に計算されて変数 p1,p2,p3 の中に入ります。4 行目のように足し合わせる事によって重ねられたグラフ p4 が出来ます。最後に『show』コマンドを使うことにより p4 が描画されます。

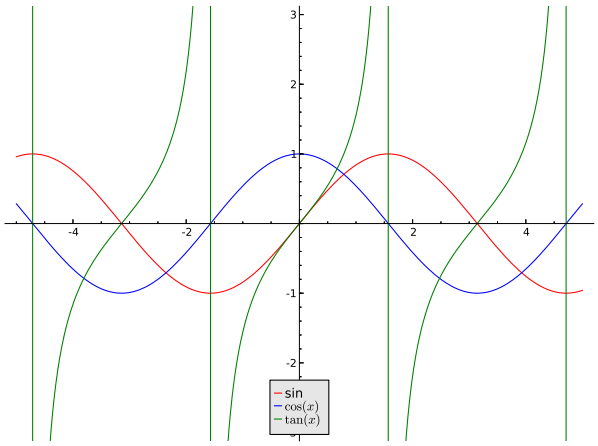


図 14　出力結果：$\sin(x), \cos(x), \tan(x)$ の色を変えて重ねて描画

グラフを重ねるのではなく，横に並べたいときは graphics_array を使います：

graphics_array——二つのグラフを横に並べて表示

例７：
```
1 p1 = plot( sin(x), (x,-4,4) )
2 p2 = plot( cos(x), (x,-4,4) )
3 p3 = graphics_array([p1,p2])
4 p3.show(aspect_ratio=1)
```

ここで aspect ratio とは縦横比のことです。横に並べるときはデフォルトだと横に縮小するため aspect ratio を 1 に指定するとよいでしょう。

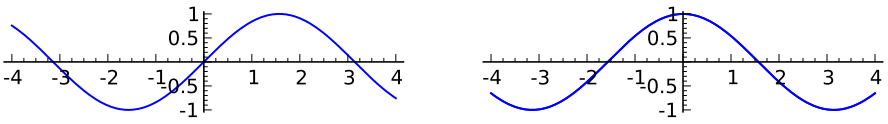


図 15　出力結果：$\sin(x), \cos(x)$ のグラフを横に並べる

`fill` コマンドを使うことによって，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：

グラフとフィリング（塗りつぶし）

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()
```

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

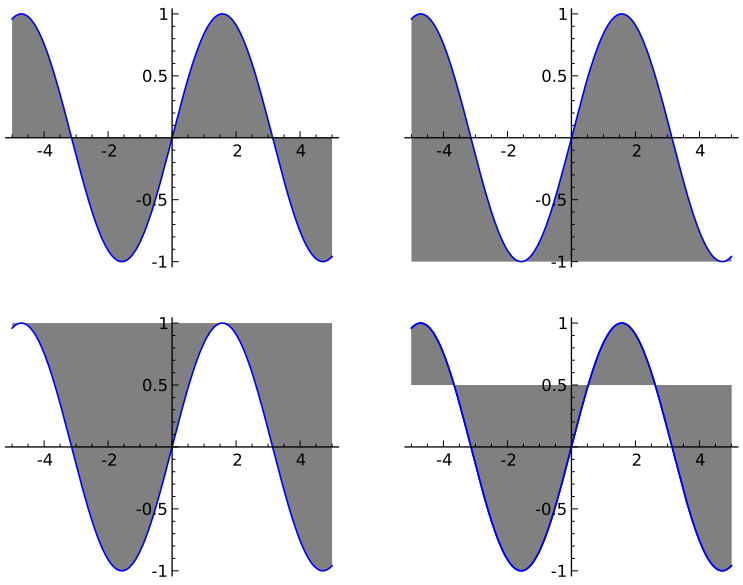


図 16　出力結果：fill の使い方

他の関数との間をフィリングにはつぎのようにします：

$\sin(x)$ のグラフを描き，関数 x^2-1 との間を色づけする

例 9：
```
1 plot( sin(x), (x,-2,2), fill = x^2-1)
```

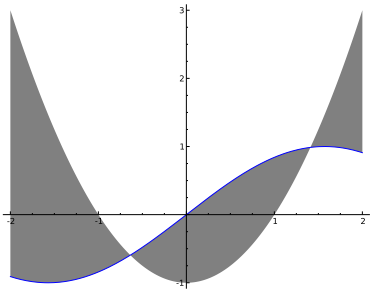


図 17　出力結果：$\sin(x)$ のグラフ，x^2-1 との間を色づけしている

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 のヘルプを表示する

```
1 plot?
```

37.4　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

implicit_plot——$f(x,y)=0$ のグラフ

```
1 var("x y")
2 implicit_plot(f(x,y), (x,a,b), (y,c,d))
```

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

ここで `x,y = var('x y')` というのは『x,y を変数として取り扱います』という宣言です。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：

$x^3+y^3-3xy=0$ のグラフ

例 **10**：
```
1 var("x y")
2 implicit_plot(x^3+y^3-3*x*y, (x,-2,2), (y,-2,2))
```

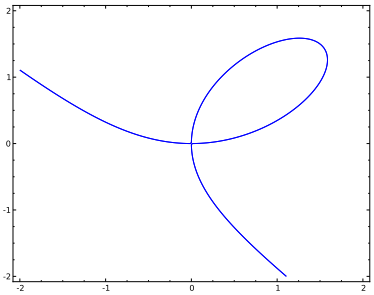


図 18　出力結果：デカルトの正葉線

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

例 **11**：
```
1 t = var('t')
2 parametric_plot([cos(t) + 2*cos(t/4), sin(t)-2*sin(t/4)],
3       (t,0, 8*pi), fill=true)
```

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。オプション `fill` を使って，囲まれる領域を色づけしています：

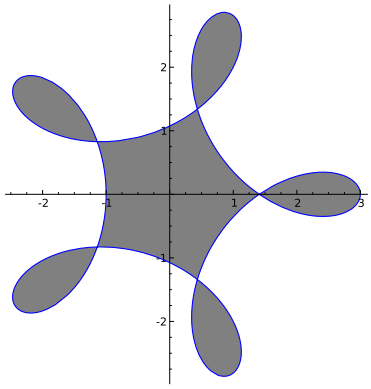


図 19　出力結果：parametric plot(with filling)

37.5　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

```
1 a = リスト
2 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]` のようなものや，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。または 3 次元のデータのからなるリストに対しては 3 次元的な描画ができます。

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)
```

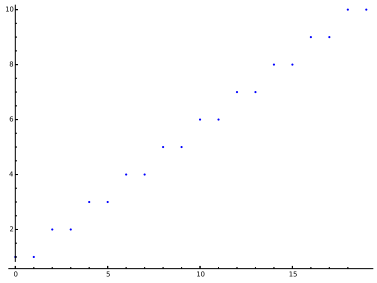


図 20　出力結果：リストのプロット

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)
```

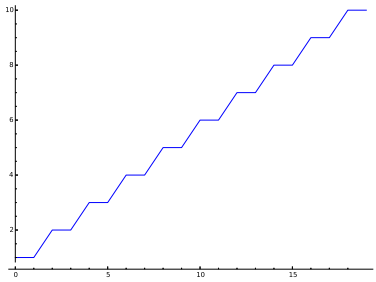


図 21　出力結果：リストのプロット (plotjoined=True)

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(i)*cos(i),sqrt(i)*sin(i)) for i in range(20)];
list_plot(a,plotjoined=True)
```

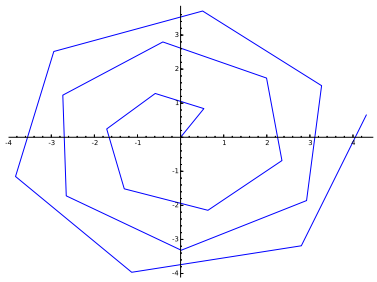


図 22　出力結果：$\{\sqrt{i}(\cos(i), \sin(i))|i = 0, 1, \cdots, 19\}$ をプロット

37.6　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

```
1 x,y = var('x,y')
2 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

例 **14**：
```
1 x,y = var('x,y')
2 region_plot(sin(x^2-y^3)>0, (x,-3, 3), (y,-3, 3))
```

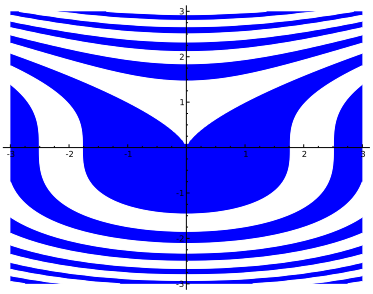


図 23　出力結果：$\sin(x^2 - y^3) > 0$

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**：
```
1 x,y = var('x,y')
2 region_plot([x>0, y>0, x^2+y^2>0.5], (x,-1,1), (y,-1,1) )
```

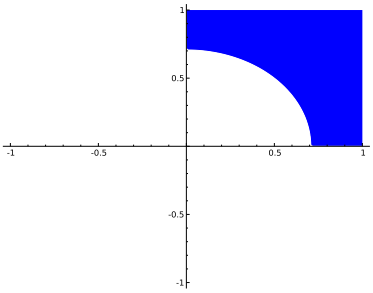


図 24　出力結果：$x > 0, y > 0, x^2 + y^2 > 0.5$ となる x, y の領域のグラフ

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ます。`region_plot` についての詳しいことはヘルプを参照してください：

```
region_plot?
```

37.7　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 などの命令と簡単に組み合わせることができる事です。

次のようなものが使えます：

- `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：

```
s1 = plot(x^2,(x,-1,1))
s2 = point((0,0), size=100, color='black')
s3 = arrow((-1/2,1/4),(1,1), color='red')
s4 = arc((0,0),0.5,sector=(0,pi), color='green')
s5 = s1 + s2 + s3 + s4
s5.show()
```

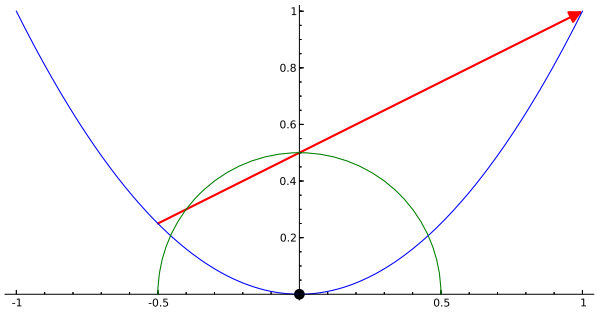


図 25　出力結果：グラフと点と矢印と円弧

38　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。

38.1　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('x 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('x y')
plot3d(sin(x-y)*y*cos(x), (x,-3,3), (y,-3,3),
     opacity=0.5, aspect_ratio=[1,1,1])
```

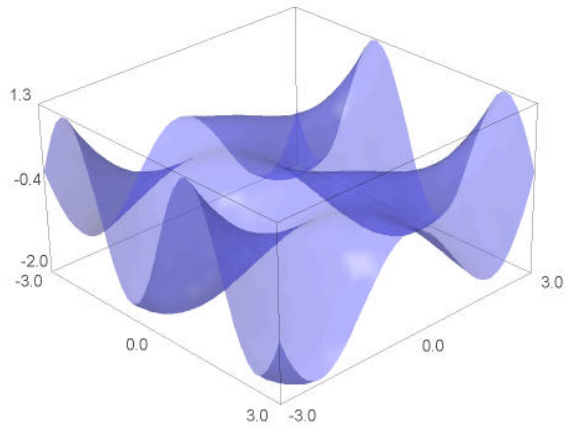


図 26　出力結果：$y\sin(x-y)\cos(x)$ のグラフ

38.2　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：
```
x, y = var('x y')
density_plot(sin(x*y), (x,-3,3), (y,-3,3))
```

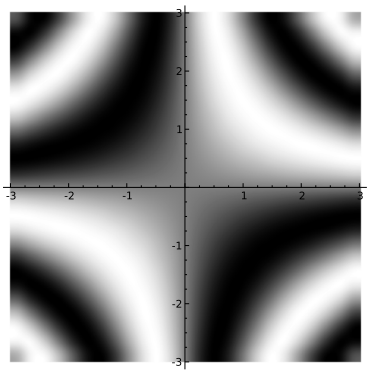


図 27　出力結果：$\sin(xy)$ の密度プロット

例 **19**：
```
1 x, y = var('x y')
2 contour_plot(cos(x)+cos(y), (x, -8, 8), (y, -8, 8),contours=20)
```

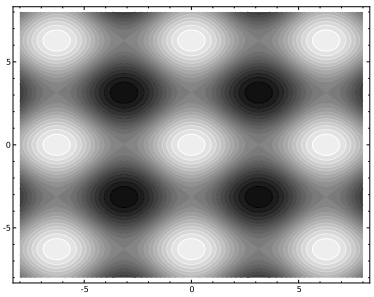


図 28　出力結果：$\cos(x)+\cos(y)$ の等高線プロット

38.3　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`

38.4　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 Sage notebook を使っていて png 形式で保存するなら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す。png 形式以外では pdf,eps,ps 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リンクが表示されます。リンクをクリックすると画像ファイルをダウンロードできます。
- 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) を実行することにより，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

```
user@debian:~$ sage plot.sage    # pdfファイルが生成される（場所は実行したディレクト
user@debian:~$
```

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。

39　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：

```
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
```

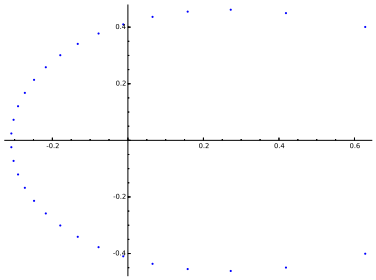


図 29　出力結果：$n = 30$ のときの零点の集合

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
```

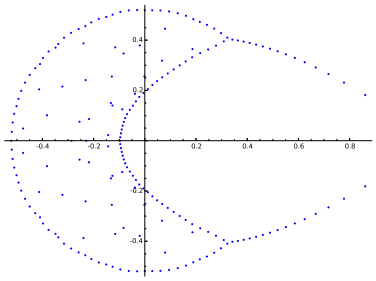


図 30　出力結果：$n = 100$ のときの零点の集合?（誤差で変な結果になる）

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)
```

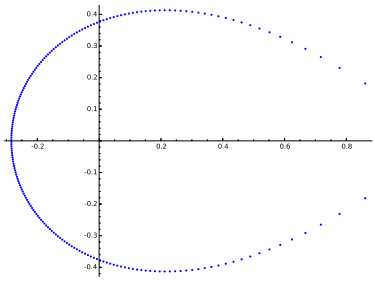


図 31　出力結果：$n = 200$ のときの零点の集合

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()
```

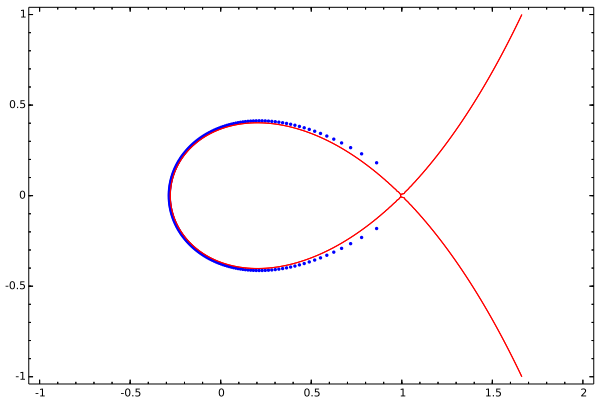


図 32　出力結果：零点の集合と $|ze^{1-z}| = 1$ のグラフを重ねる

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)
```

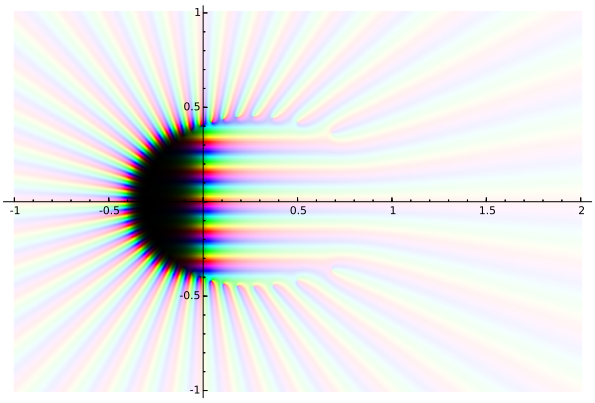


図 33　出力結果：複素関数 $\sum_{k=1}^{40}(40z)^k/k!$ の形

40　練習問題

上の 3 節の解説にあるグラフの描画を試してみよう。グラフが描画された SageMath のワークシートを提出すること。ファイル名は `sagews02.sws` とすること。

41 代数的計算

41.1 文字式の展開・因数分解・単純化

関数の展開 (expand)

f を関数としたとき f の展開は次で行う：

```
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する：

```
var('x');     f = (x+1)^3
print f.expand()
```

実行結果の例

```
x^3 + 3*x^2 + 3*x + 1
```

関数 f の因数分解 (factor)

関数 f の因数分解は次で行う：

```
 f.factor()      または     factor(f)
```

試しに $x^3+3x-2x-6$ の因数分解をしてみましょう。

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

実行結果の例

```
(x + 3)*(x^2 - 2)
```

となります（整数係数の範囲でしか因数分解されません）。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には simplify があります。

関数 f の単純化 (simplify と simplify_full)

複雑な関数 f を単純化して整理するには

```
f.simplify()     または     simplify(f)
```

とします。時間はかかるけれどもできる限り単純にするには次のようにします：

```
f.simplify_full()    または    simplify_full(f)
```

例として $f=x^2+2x+5-3x$ を単純化してみましょう：

```
f = x^2 + 2*x + 5 - 3*x
simplify(f)
```

実行結果の例

```
x^2 - x + 5
```

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: var('x');    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
```

41.2 $x^{105}-1$ 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ます：

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

答えは

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

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。

```
var('x')
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 が出てきます。

41.3 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。

式 f の部分分数分解

```
f.partial_fraction()
```

例：

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

41.4 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

$$\begin{aligned}\frac{345}{29} =& 11 + \frac{26}{29} = 11 + \cfrac{1}{\cfrac{29}{26}} = 11 + \cfrac{1}{1 + \cfrac{3}{26}} = 11 + \cfrac{1}{1 + \cfrac{1}{\cfrac{26}{3}}} \\ =& 11 + \cfrac{1}{1 + \cfrac{1}{8 + \cfrac{2}{3}}} = 11 + \cfrac{1}{1 + \cfrac{1}{8 + \cfrac{1}{\cfrac{3}{2}}}} \\ =& 11 + \cfrac{1}{1 + \cfrac{1}{8 + \cfrac{1}{1 + \cfrac{1}{2}}}}\end{aligned}$$

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を決める bits=40 などを指定します。

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, 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。

41.5 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。

式 f の極限 $\lim_{x\to a} f(x)$

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

上の命令で f.limit(x=oo) 部分は limit(f, x = oo) と書いても同じ事です。

41.6 代数的な和の計算

Sage で和を計算するには sum を使います。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

```
1 var('k')              # kを変数とする
2 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。

まずは $1+2+\cdots+50$ を計算してみましょう。

```
1 var('k');     sum(k,k,1,50)
```

実行結果の例

```
1275
```

sum は数値としての和の算出だけでなく

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

のように厳密に成り立つ式も導出してくれます：

```
1 sage: var('k m')                  # kとmを変数とする
2 sage :ans = sum(k,k,1,m)          # 和を計算してansに代入
3 sage: print ans                   # ansをプリントする
4 1/2*m^2 + 1/2*m           # これが答え
5 sage: factor(ans)         # ansを因数分解
6 1/2*(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：

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

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

実行結果の例

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

42　方程式の解

42.1　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

```
1 >>> x = 3
2 >>> print x
3 3
```

と書けば変数 x が新たに作られて，x には 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

```
var('x y')
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。

```
var('x y')                    # x, y を不定元(=形式的文字)とする。
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：

```
print f              #fをプリント
print f.rhs()        #fの右辺をプリント
print f.lhs()        #fの左辺をプリント
```

実行結果の例

```
x^2 + y^2 == x*y + 1
x*y + 1
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。

42.2 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 を使います：

方程式 $f(x) = g(x)$ を解く

```
var('x')
solve(f(x)==g(x),x)      # f(x)=g(x)をxについて解く
```

方程式の等号は『==』であることに注意

試しに $2x + 3 = 7$ を解いてみましょう：

```
var('x')
solve(2*x+3==7,x)
```

実行結果の例

```
[x==2]
```

変数 x, y についての連立方程式を解くには次のようにします：

連立方程式の解を求める (solve)

```
var('x y')
solve([方程式1,方程式2],x,y)
```

例えば連立方程式 $x + y = 6$, $x - y = 4$ を解くには次のようにします：

```
var('x y')
solve([x+y==6, x-y==4], x,y)
```

実行結果の例

```
[[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：

```
var('x'); solve(x^3+2*x+1==0)
```

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)]
```

結果は複雑で少し見づらいですが，解はリストの形をしているので好きな部分を取り出すことができます。たとえば，3つの解を改行してプリントするには：

```
a = solve(x^3+2*x+1==0,x)
print a[0]; print ""
print a[1]; print ""
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) - 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)
```

1番目の解の値だけが必要な場合には，方程式の右辺を取り出せばよいので `rhs()` を使って

```
a = solve(x^3+2*x+1==0,x)
a[0].rhs()
```

実行結果の例

```
-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)
```

`solve` コマンドでは上のような3次方程式や4次方程式は代数的に解くことが可能ですが，5次以上の方程式は代数的に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：

```
solve(x^5+3*x+1==0,x)
```

実行結果の例

```
[0=x^5+3x+1]
```

でも特別な係数をもつ方程式については代数的に解くことができるかもしれません：

```
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：

```
solve(sin(x)==1/2,x)
```

実行結果の例

```
[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：

```
1 solve(sin(x)*cos(x)==1/2,x)
```

実行結果の例

```
[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くあります。

42.3　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

```
1 find_root(方程式,a,b)
```

find_root は区間の中に解が複数個存在しても 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：

```
1 sage: find_root(x^2+2*x-9==0,-5,5)
2 2.1622776601683791
```

上と同じ方程式で解を探す範囲を狭くすると，解が無くなってしまいますが，このときはエラーとなります：

```
1 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，答えは厳密ではないけれども，複雑な関数であっても数値解を求める事ができる点にあります：

```
1 sage: find_root(tan(x)==8-x^3,-10,10)
2 2.1261689044831993
```

ただ，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：

```
1 sage: find_root(x^400,-5,5)
2 0.13155619723565876
```

もちろん上の方程式の解は $x = 0$ のみです。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。

右極限と左極限の指定は次の様にします：

```
1 sage: limit(1/x, x=0, dir='+')    # 右極限
2 +Infinity                         # 正の無限大
3 sage: limit(1/x, x=0, dir='-')    # 左極限
4 -Infinity                         # 負の無限大
```

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

$$\lim_{x\to 0} x^a \tag{8}$$

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?            #a の値を聞いてくる
```

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?                               #まだ値は定まらない。
```

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
```

43　Sage による微分と積分

関数 $f(x)$ の微分は次で行います：

関数 $f(x)$ の微分

```
diff(f,x)　または　derivative(f,x)　または　f.derivative(x)
```

関数 f の x についての微分。

例：

```
sage: diff(sin(x),x)
cos(x)
```

Sage による不定積分は次の命令で行います：

関数 $f(x)$ の不定積分 $\int f(x)dx$

```
integral(f(x),x)　または　integrate(f(x),x)
```

代数的に計算され，結果は厳密です。ただし出力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)
```

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：

```
a = var('a')
integral(cos(a*x),x)
```

実行結果の例

```
sin(a*x)/a
```

関数 $f(x)$ の定積分 $\int_a^b f(x)dx$

x の関数 f の定積分は

```
integral(f,x,a,b)　または　integral(f,(x,a,b))　または　f.integral(x,a,b)
```

integral のかわりに integrate を使っても結果は同じです。

例：

```
sage: integral(x^2,x,0,2)
```

```
8/3
```

43.1 不定積分はできないけど定積分なら出来る場合

定積分は求まらないが，不定積分の厳密値なら得られる場合があります。たとえば $(1-x^4)^{-1/2}$ の原始関数は integral では求まりませんが，定積分 $\int_0^1 (1-x^4)^{-1/2}dx$ は求まります。次を実行してみましょう：

```
var('x')
ans = integrate(1/sqrt((1-x^4)),0,1)      # 積分値をansとする
print ans          #ansをプリント
print n(ans)       #ansの数値をプリント
```

実行結果の例

```
1/4*beta(1/4, 1/2)
1.31102877714606
```

ここで beta はベータ関数

$$\beta(x,y) = \int_0^1 t^{x-1}(1-t)^{y-1}dt$$

です。Sage4*ではベータ関数はデフォルトでサポートされていなかったので数値を得るためには Maxima を呼び出す必要がありましたが，Sage5 からはサポートされて手軽に扱えるようになりました。

43.2 数値積分

integrate 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。そのような場合でも数値積分の命令 numerical_integral を使うことで，定積分の近似値を得ることができます。

Sage による数値積分は次の命令で行います：

関数 $f(x)$ の数値積分 $\int_a^b f(x)dx$

```
numerical_integral(f(x),a,b)
```

結果として（値，誤差）が返されます。オプション max_points により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：

```
sage: numerical_integral(sin(x),0,pi)
(1.9999999999999998, 2.220446049250313e-14)
```

上の積分は厳密には 2 となるはずですが，微少の誤差が出ています。実行結果の二つ目は誤差を表していて，ここでは積分値はおよそ $2*10^{-14}$ の誤差が出ていることを表しています。ただし，これは厳密な誤差ではなくアルゴリズムから予測される真の値との誤差です。数値だけほしい場合には，実行結果の第 0 成分だけ取り出せばよいので，次のようにします：

```
sage: val = numerical_integral(sin(x),0,pi)
sage: val[0]
1.9999999999999998
```

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特性によります。振動が激しい関数や，積分が緩やかに発散する関数の場合などは，真の値と大幅に違った値になってしまうこともあります。

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。例えば，次の数値積分は誤差が 0.0001 ほどありますが，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：

```
sage: numerical_integral(sin(1/x),0,1)
(0.5040702199680797, 0.00012692441400448108)
sage: numerical_integral(sin(1/x),0,1,max_points=1000)
(0.5040670267347616, 1.4633387576762725e-05)
```

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，maxima という数式処理ソフトを用いて数値積分を行う命令があります。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**）となります。

エラーコードは 0 から 6 まであり

- 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

を意味します。次の例を実行してみましょう：

```
sage: f=sin(1/x);   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=sin(1/x);     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 0.01 の精度で確からしい値が得られたことがわかります。

44 Taylor 展開

44.1 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

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-1) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n + \cdots$$

の右辺の事をいいます。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

```
var('x')
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('x y')
taylor(sin(x+y),(x,0),(y,0),4)
```

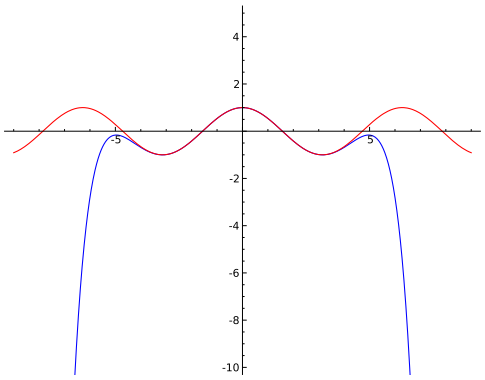
の実行結果は次のようになります：

$$-\frac{1}{6}x^3 - \frac{1}{2}x^2y - \frac{1}{2}xy^2 - \frac{1}{6}y^3 + x + y$$

44.2 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：

```
var('x')              # xを変数とする
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
```



45　練習問題

以下の問題の答えを Sage を使って求めてみましょう。

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。

$$(8)\quad \text{黄金比} = \texttt{golden\_ratio} \qquad (9)\quad \text{円周率} = \texttt{pi}$$

次の関数を x で微分せよ。

(10) $\sin(cx)$（c は定数）

(11) $|x|/(1+x^2)$(絶対値 $|x|$ は `abs(x)` で表します。(absolute value の略))

次の x についての関数の原始関数を求めよ。

$$(12)\quad \frac{1}{a+x} \qquad (13)\quad a^x \qquad (14)\quad \frac{1}{\sqrt{a^2+x^2}}$$

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。

(15) x^2，区間 $[0,1]$

(16) $\sin(x)$, 区間 $[0,\pi]$

(17) $x\cos(x)$，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。

(18) x^2e^x

(19) $\sin(x)/(1+x^2)$

上の問題の答えをすべて出力する Sage プログラムを作成しましょう。ファイル名は `answer.sage` とすること。提出するプログラムはコマンドラインから `sage answer.sage` と実行したときに問題番号とその解答

を表示 (print) するようにせよ。例えば，最初の問題の答えは次のようになる：

- ファイル名：answer.sage

```
# -*- coding: utf-8 -*-
print '(1) ',
var('x h')
f1 = ((x+h)^3-x^3)/h
print limit(f1,h=0)
```

46　微分方程式

46.1　常微分方程式とベクトル場

微分方程式

$$\frac{dy}{dx} = y - x \tag{9}$$

を考える。これの一般解は，$y(x) = x + 1 + Ce^x$ である。ここに C は任意定数。初期値問題

$$\frac{dy}{dx} = y - x, \qquad y(0) = \frac{2}{3} \tag{10}$$

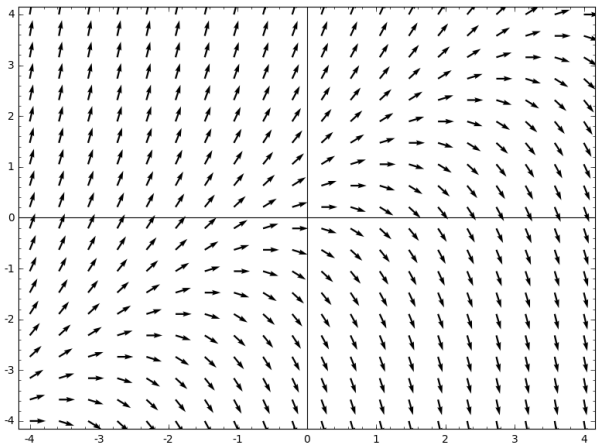
を考える。上の一般解に $x = 0$ の値を代入して $y(0) = 1 + C = 2/3$ から C を求めると，$C = -1/3$ となることがわかる。よってこの初期値問題の解は

$$y(x) = x + 1 - \frac{1}{3}e^x \tag{11}$$

である。このようにして代数的な計算によって微分方程式の解を求めることができるが，コンピューターによる描画を用いると解の様子を視覚的に観察することができる。

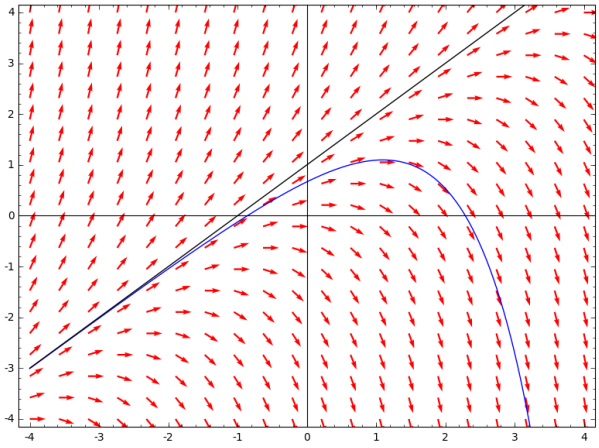
微分方程式 (9) の解曲線 $y = y(x)$ は平面上の点 (x, y) で傾き $y - x$ を持つ。そこで，点 (x, y) に $(1, y - x)$ の向きの単位ベクトルが生えているベクトル場を描いてみよう。そのようなベクトル場は勾配場 (slope field) と呼ばれる。つまり，$m = (1 + (y - x)^2)^{-1/2}$ として，ベクトル場 $V(x, y) = (m, m(y - x))$ を描けばよい。

```
var('x 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 (9) 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たどることで得られる。上の絵を見れば，解の振る舞いは一目瞭然であろう。具体的には直線 $y = x + 1$ を境に，その上と下で $x \to \infty$ のときの振る舞いは異なる。この勾配場 (赤) に $y = x + 1$ の直線 (黒) および解 (11)(青) を重ねてみよう。

```
var('x 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))
asym = plot(x+1, (x,-4,4), color='black')
(vecf+sol+asym).show(ymin=-4, ymax=4)
```



46.2 Euler 法

微分方程式の多くは代数的に解くことはできない。しかし，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

$$\frac{dy}{dx} = f(x,y) \tag{12}$$

及び初期条件 $y(x_0) = y_0$ を考える。(x_0, y_0) は与える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

$$y(x+dx) = y(x) + f(x,y)dx \tag{13}$$

と書くことができる。これは y の点 x での値から無限小だけ進んだ点 $x+dx$ での値を決める方程式と考えられる。Euler 法はこれに基づいて $y(x)$ を近似する方法である。Δx を小さい数とし，$x_1 = x_0 + \Delta x$ での y の値は

$$y_1 = y_0 + f(x_0, y_0)\Delta x \tag{14}$$

で近似される。$x_2 = x_1 + \Delta x$ での y の値は

$$y_2 = y_1 + f(x_1, y_1)\Delta x \tag{15}$$

で近似される。同様にして $x_n = x_0 + n\Delta x$ での y の値は

$$y_n = y_{n-1} + f(x_{n-1}, y_{n-1})\Delta x \tag{16}$$

によって近似される。

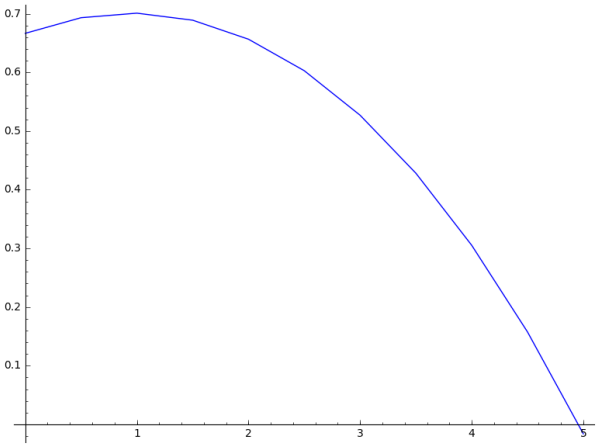
Euler 法に従って微分方程式の初期値問題 (10) の解 $y(x)$ を近似してみよう。以下では，Δx と n の値を適当に指定して，リスト $((x_0,y_0),(x_1,y_1),\cdots,(x_n,y_n))$ を作る。

```
x0, y0 = 0, 2/3    # 初期条件をセット
Dx= 0.5            # Delta xを設定
var('x 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

```
[(0, 2/3), (0.500000000000000, 0.693333333333333), (1.00000000000000,
0.701066666666667), (1.50000000000000, 0.689109333333333),
(2.00000000000000, 0.656673706666667), (2.50000000000000,
0.602940654933333), (3.00000000000000, 0.527058281130667),
(3.50000000000000, 0.428140612375893), (4.00000000000000,
0.305266236870929), (4.50000000000000, 0.157476886345766),
(5.00000000000000, -0.0162240382004033)]
```

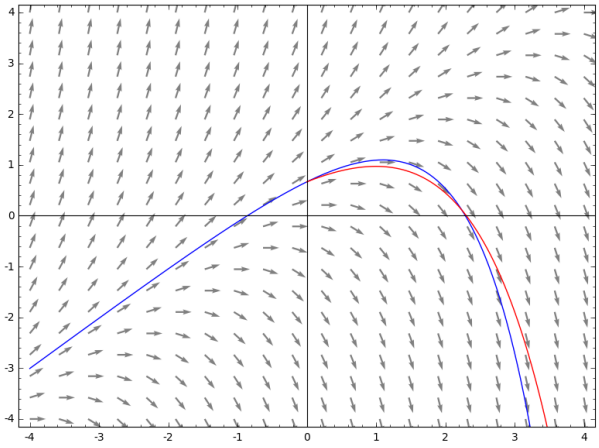


Euler 法による近似解と真の解とを比較してみよう。

```
var('x 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。）

46.3 微分方程式の厳密解

この節では `desolve` 関数を使って微分方程式を解きます。`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するときに設定します
- `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。

これらのオプションは省略可能で，指定しない場合 `False` になります。

さて簡単な微分方程式

$$y'(x) = y(x) \tag{17}$$

を解いてみましょう。これ解くには次のようにします：

```
1 var('x')                      # x は形式的な変数
2 y = function('y', x)          # y を x の関数とする!!
3 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。

```
var('x');    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
```

46.3.1 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。

$$y'(x) + 2y(x)\sin(x) = 0 \tag{18}$$

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

$$\frac{y'}{y} = -2\sin(x) \tag{19}$$

と変形して[*14]，両辺を x で積分する事により

$$\log y = \int \frac{y'}{y} dx = 2\cos(x) + c \tag{20}$$

となります。これを y について解くことにより答えは

$$y(x) = \exp(2\cos(x) + c) \tag{21}$$

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，(18) の解は

$$y(x) = Ce^{2\cos(x)} \tag{22}$$

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：

```
x = var('x');    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))
```

46.3.2 微分方程式の解法 2

次の微分方程式を考えます：

$$y'(x) = \frac{x - y(x)}{x + y(x)}. \tag{23}$$

これも変数分離型として解く事ができますが，Sage に解かせてみましょう：

[*14] $y(x) = 0$ 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。

```
x = var('x');  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 (23) の解 $y(x)$ は，微分を含まない方程式

$$-\frac{1}{2}x^2 + xy(x) + \frac{1}{2}y(x)^2 = C \tag{24}$$

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：

```
x = var('x');  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。

± と定数 C 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
```

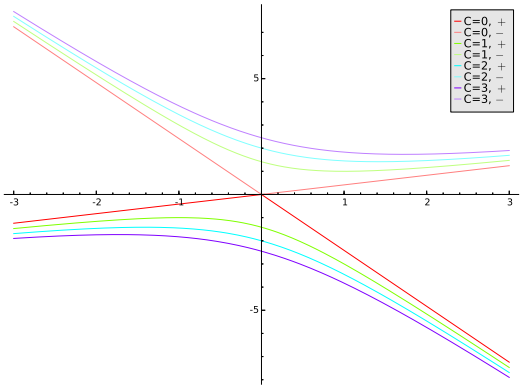


図 34　出力結果：微分方程式 (23) の解のグラフ

46.3.3　微分方程式の解法３

次の微分方程式を考えます：

$$y' = \sin(x + y) + \sin(x - y) \tag{25}$$

これも変数分離型で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，

```
var('x');   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.
```

右辺を加法定理で因数分解してやれば解く事ができます：

```
var('x');  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)
```

46.4　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

$$y''(x) - 8y'(x) + 15y(x) = 0 \tag{26}$$

を解いてみましょう。

```
x = var('x');  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。

46.5　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：

```
var('x');  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。

46.6　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で独立変数を `ivar=x` と指定してやります。

```
var('x a')      # a, x 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('x 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('x 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)
```

46.7　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

$$y(x_0) = y_0 \tag{27}$$

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。

```
var('x');   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

```
x = var('x');   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[*15]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

```
var('x');    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。

46.8　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：

```
var('x');    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：

```
var('x');    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。

47　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。

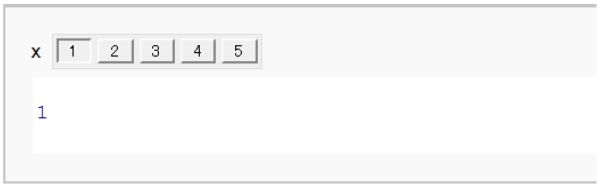
簡単な例を使って@interact の使い方を紹介します。関数 f(x) を，引数 x に対して x^2 を画面に表示する関数である，と定義します：

```
def f(x):          # 関数 f を定義
    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：

[*15] 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。

```
@interact          # インタラクティブな操作をするためのおまじない
def f(x=[1,2,3,4,5]):    # 関数の中で引数の取る値を指定
    print x^2
```



番号付きのボタンをクリックすると，その下の実行結果が変わります。プログラムの中のリスト `[1,2,3,4,5]` がボタンの番号に対応しています。

上のものは最も簡単なインタラクティブな操作ですが，一般には次のような手順によって行います：

@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（例 `x=(1,2,3,4,5)`）

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：

- リスト：`x=[1,2,3,4,5]`　(ボタン)
- タプル：`x=(1,2,3,4,5)`　(スライダー)
- 1 から 5 までの自然数：`(1..5)`　(操作はスライダー)
- 0.0 から 10.0 までの連続的な実数：`x=(0,10)`　(操作はスライダー)
- 5 から 10 までの 0.2 刻みの実数：`(5,10,0.2)`　（操作はスライダー）
- 1 から 8 までの実数をとるスライダーで，初期位置は 5　⇒　`slider(1,8,default=5)`
- 文字を入力する `input_box(type=str)`

`slider` と `input_box` はオプションを次のように指定するすることができる：

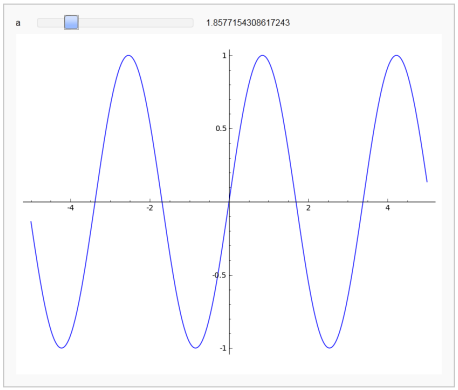
- `input_box(default = "sin(x)",label = "hoge", type=str)`
- `slider(0,1,.01,.5,label = 'start value')`

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 までの実数になります。

```
@interact
def f(x=(-5,5)):
    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：

```
@interact
def f(a=(1,5)):                 #  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
```



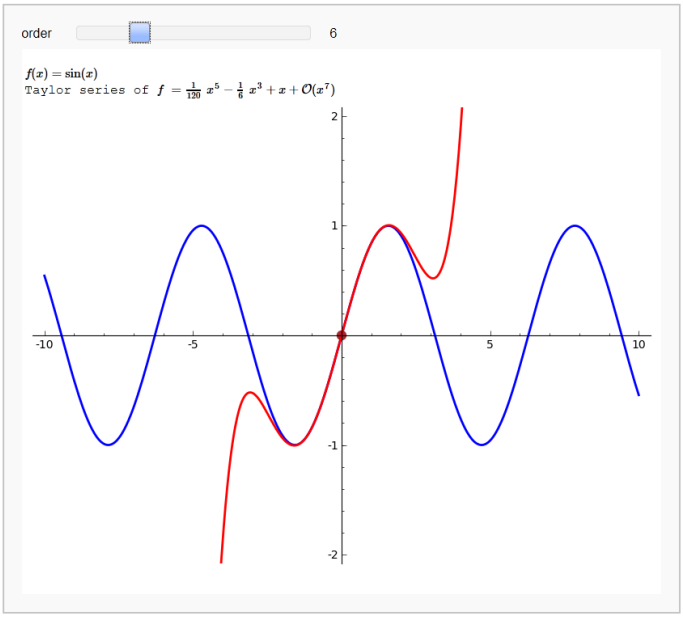
$\sin(x)$ とテイラー級数を同時に描いてみましょう：

```
var('x')
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。

```
var('x')
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。

次のようにすると，関数 sin(x) も固定したままではなく変化させることができます：

```
1  var('x')
2  x0 = 0
3  @interact
4  def show_taylor(func=input_box(default="sin(x)", label="function", type=str),
5                  order=(1..20), xmin=(-5,5), xmax=(0,5)):
6      f(x) = eval(func)                    # 文字列funcを式に直したものがf(x)
7      dot = point((x0,f(x=x0)),pointsize=80,rgbcolor='brown')  # 点
8      p = plot(f(x),x,xmin,xmax, thickness=2)                  # fをプロット
9      ft = f.taylor(x,x0,order)                        #ftはfのTaylor展開
10     pt = plot(ft,xmin,xmax, color='red', thickness=2)  #ftをプロット
11     pretty_print(html('$f(x) = %s$'%latex(f)))
12     pretty_print(html('Taylor series of $f$ $= %s+\mathcal{O}(x^{%s})$'
13                       %(latex(ft),order+1)))
14     show(dot + p + pt, ymin = -2, ymax = 2)    # グラフを表示
```

@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

48　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)]
```

```
b = animate(lis1, figsize=[3,2])
b.show()
b.show(delay=2)
b.show(delay=3)
```

描画には計算時間が多少かかります。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。

48.1 高木関数の描画

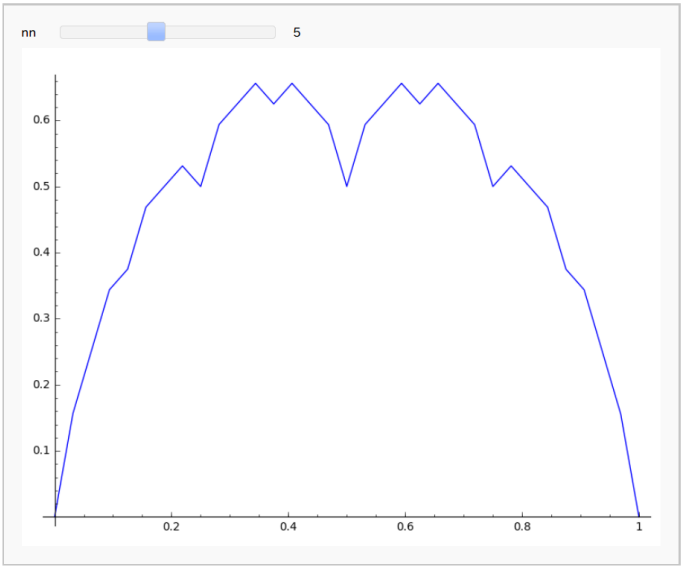
高木関数とは

$$T(x) = \sum_{n=0}^{\infty} \frac{s(2^n x)}{2^n} \tag{28}$$

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))

@interact
def plotTakagi(nn=(1..10)):
    f = lambda x: sum( [s(2^k*x)/2^k for k in range(nn)] )
    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。

```
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)
```

48.2 練習問題

何か面白い gif 動画を生成する Sage プログラムを作ってみましょう。

例えば高木関数のグラフ (takagi.gif) を生成する Sage プログラムは次のようになります。

- ファイル名：**GenTakagi.sage**

```
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)]
pp = animate(figs)
pp.gif(savefile='./takagi.gif', delay=70)
```

上のファイルを端末から

```
$ sage GenTakagi.sage
$
```

と実行すると，実行したフォルダに takagi.gif が作られます。

49　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。

49.1　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) $\begin{pmatrix}1\\2\end{pmatrix}$ = `vector([1,2])`

(2) $\begin{pmatrix}1&2&3\\4&5&6\end{pmatrix}$ = `matrix([[1,2,3],[4,5,6]])`

(3) $\begin{pmatrix}1&2&3\\4&5&6\end{pmatrix}$ = `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：

```
1 a = vector([1,2])      # aをベクトル(1,2)とする
2 print a
```

実行結果の例

```
(1, 2)
```

```
1 b = matrix([[1,2],[3,4]])  # 行列bを定義
2 print b
```

実行結果の例

```
[1 2]
[3 4]
```

```
1 c = matrix(2,3,[1,2,3,4,5,6])    # 行列cを定義
2 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。

```
1 def elem(i,j):       # 関数elemの定義
2     return i+j^2     # i+j^2を返す
3
4 aa = matrix(5,5, elem)     # i,j成分が elem(i-1,j-1) となる行列
```

```
5 print aa
6 print aa[3,4]
```

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    # aa の 4 行 5 列目の成分
```

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。

49.2　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 にやらせるには

```
1 print 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：

```
1 print 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 となる。複素共役はとらない。
```

同様に行列の積は次のように計算します：

```
1 print matrix([[1,2],[-4,2]]) * matrix([[2,-1],[-3,2]])
```

実行結果の例

```
[ -4   3]
[-14   8]
```

49.3　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)`	A^t
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。

では，実際に上の命令を試してみましょう。

```
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()      # Aの転置行列
print ''    # 改行
print det(A)             # Aの行列式
print ''    # 改行
print A.trace()          # Aのトレース
print ''    # 改行
print A^-1               # Aの逆行列
```

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()                 # nullity
```

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 lambda 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。

49.4　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。例えば，前者は数学的興味のために行われ，後者は応用上のために行われます。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— を対象としていても，計算上は異なる取り扱いをしなければなりません。一般的に，厳密な計算には時間がかかるため，近似的な数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。さいわい，Sage の行列計算では，簡単な指定をするだけで，厳密な計算か数値計算かを区別して計算を行わせることができます。

Sage で行列計算の固有値や固有ベクトル，ジョルダン標準形などを計算させるときには，どこの係数環（または体）で計算をするのか意識する必要があります。行列の成分がどの環・体であるかによって計算できること，できないこと が異なってきます。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) を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[16]。

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：

```
1 mat = matrix(2,2,[1,2,3,4])       # 行列を定義
2 print 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：

```
1 mat = matrix(2,2,[1/2,2,3,4])     # 行列の要素に有理数1/2がある
2 print 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：

```
1 mat = matrix(SR, [[1,2],[3,4]])    # 行列をSR上で定義
2 print 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：

```
1 a = matrix([[1,2],[3,4]])          # 行列aを定義
2 print a.base_ring()                # 行列aのbase ringをプリント
3 a = matrix(SR, a)                  # aのbase ringをSRに変更
```

[16] 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

```
4 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
```

49.5 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()`	$\det(xI - A)$	多項式の変数は `x`
固有値	`A.eigenvalues()`	$\det(\lambda I - A) = 0$ 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

49.6 固有値・固有ベクトル

正方行列 A があるベクトル v と定数 λ に対して

$$Av = \lambda v, \quad (v \neq 0)$$

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 $(A - \lambda I)v = 0$ 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 $\det(A - \lambda I) = 0$ であることがわかります。つまり多項式 $\det(A - \lambda I) = 0$ の解が固有値です。Sage で固有値をする場合，`base_ring` の設定によって答えの表示が異なります。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 で計算します。

```
1 mat = matrix(SR, 2,2,[1,2,3,4])     #base_ringはSRに設定
2 print 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]
```

一つ目の結果では厳密な値が表示されました。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
```

となって，固有値を具体的に決定できなかったことが分かります[*17]。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[*18]。

そこで，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：

```
mat = matrix(CDF,mat)     # base_ringをCDFに変更
print 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。

```
mat2 = matrix(SR, 2,2,[1,2,3,4])     # base_ringはSRに設定
eigv = mat2.eigenvectors_right()      # 固有ベクトルの組を定義
print eigv
```

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)
```

49.7　対角化とその応用

49.7.1　行列の対角化

行列の対角化は，行列の冪や指数関数を計算するために必要な手順の一つです。行列 A を対角化するとは，ある正則行列 P を見つけて

$$P^{-1}AP = \mathrm{diag}[\lambda_1, \cdots, \lambda_n]：\text{対角行列}$$

を計算することです。通常，対角化行列 P を作るためには，A の固有ベクトルを計算すればよいのですが，Sage では，eigenmatrix_right() で対角化を簡単に行うことができます。

[*17] 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。
[*18] もちろん結果は誤差を含む近似値になりますが。

行列 $\begin{pmatrix} 1 & 4 \\ 2 & 3 \end{pmatrix}$ を対角化してみましょう。

```
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。

49.7.2　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

$$\exp(tA) = \sum_{n=0}^{\infty} \frac{t^n}{n!} A^n, \qquad t \in \mathbb{C} \tag{29}$$

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 $(x(t), v(t))^t$ 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。

49.7.3 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 $a_1 = 1, a_2 = 1,$

$$a_{n+2} = a_{n+1} + a_n, \qquad n = 1, 2, 3, \cdots \tag{30}$$

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

$$\begin{pmatrix} a_{n+1} \\ a_n \end{pmatrix} = A \begin{pmatrix} a_n \\ a_{n-1} \end{pmatrix} = A^2 \begin{pmatrix} a_{n-1} \\ a_{n-2} \end{pmatrix} = \cdots = A^{n-1} \begin{pmatrix} a_2 \\ a_1 \end{pmatrix} = A^{n-1} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \tag{31}$$

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 = ', P,  ''
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

$$A^n = P \begin{pmatrix} (\frac{-1+\sqrt{5}}{2})^n & 0 \\ 0 & (\frac{\sqrt{5}+1}{2})^n \end{pmatrix} P^{-1} \tag{32}$$

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

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} \tag{33}$$

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。

フィボナッチ数列に自然数しか現れないのに，その一般項を表すには平方根が必要であるというのは面白い事実です。

49.8 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。

49.9 練習問題

49.9.1 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

$$Q^+(g):=\begin{pmatrix} 1 & \sqrt{1}g & 0 & 0 & 0 & 0 & 0 & 0 \\ \sqrt{1}g & 0 & \sqrt{2}g & 0 & 0 & 0 & 0 & 0 \\ 0 & \sqrt{2}g & 3 & \sqrt{3}g & 0 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{3}g & 2 & \sqrt{4}g & 0 & 0 & 0 \\ 0 & 0 & 0 & \sqrt{2}g & 5 & \sqrt{5}g & 0 & 0 \\ 0 & 0 & 0 & 0 & \sqrt{5}g & 4 & \ddots & \\ 0 & 0 & 0 & 0 & 0 & \ddots & \ddots & \sqrt{N}g \\ 0 & 0 & 0 & 0 & 0 & & \sqrt{N}g & N+(-1)^N \end{pmatrix}+g^2I$$

$$Q^-(g):=\begin{pmatrix} -1 & g & 0 & 0 & 0 & 0 & 0 & 0 \\ g & 2 & \sqrt{2}g & 0 & 0 & 0 & 0 & 0 \\ 0 & \sqrt{2}g & 1 & \sqrt{3}g & 0 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{3}g & 4 & \sqrt{4}g & 0 & 0 & 0 \\ 0 & 0 & 0 & \sqrt{4}g & 3 & \sqrt{5}g & 0 & 0 \\ 0 & 0 & 0 & 0 & \sqrt{5}g & 6 & \ddots & \\ 0 & 0 & 0 & 0 & 0 & \ddots & \ddots & \sqrt{N}g \\ 0 & 0 & 0 & 0 & 0 & & \sqrt{N}g & N-(-1)^N \end{pmatrix}+g^2I$$

を定義する。I は単位行列。$Q^{\pm}(g)$ 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。すべてのグラフを足し合わせて一つのグラフにする（固有値ごとに色を変えておくとよい）[*19]。

49.9.2 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

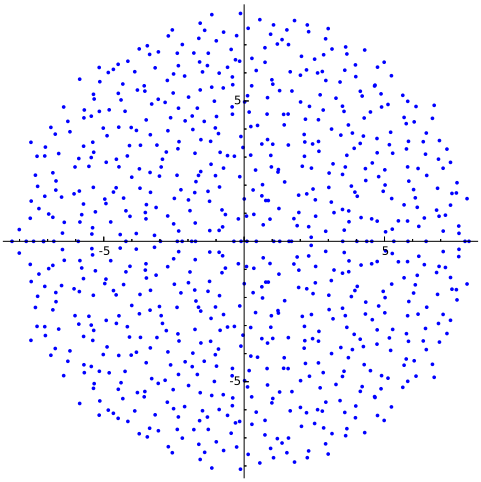
[*19] 注：このような行列は 2 準位を持つ原子とレーザー光との相互作用を表すモデルのハミルトニアンを記述しており Rabi モデルと呼ばれる。対応する固有値はその量子系のエネルギー準位を表している。

ンガー作用素の固有値や固有ベクトルの挙動は絶縁体のメカニズムの数学的な説明を与える。

***

問題 $-1/2$ から $1/2$ までのランダムな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[*20]。

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 $(0, 1)$ 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,
```

実行結果の例

```
-0.0357016806704 0.0999712876727 -0.169974222529 0.39728018763
```

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。

[*20] 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。

50　乱数

乱数とはさいころの目のようにでたらめ[*21]に表れる数字のことです。乱数は，乱雑[*22]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。決定的な手続きによって作られる，乱数を近似するような数の列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[*23]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。

50.1　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。[*24]

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

[*21] 出鱈目
[*22] 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
[*23] 数学的なというより思想上の問題
[*24] $[0.0, 1.0)$ の中にある浮動小数点数が無作為に抽出されます。

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が難しくなるでしょう[*25]。

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[*26]。

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。

[*25] 幽霊の目撃情報のように

[*26] 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

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：

```
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()
```

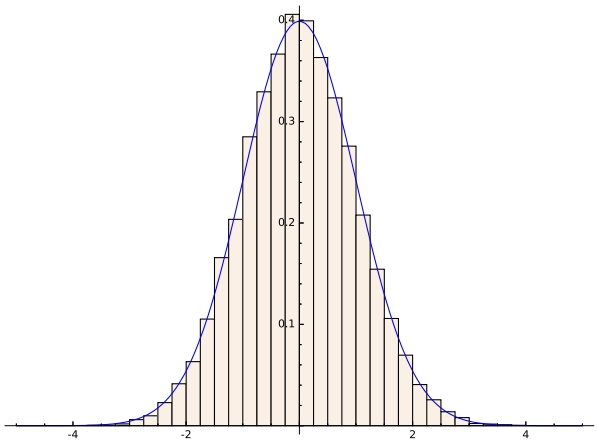


図 35　ガウス分布のヒストグラム

50.2　モンテカルロ法

50.2.1　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。

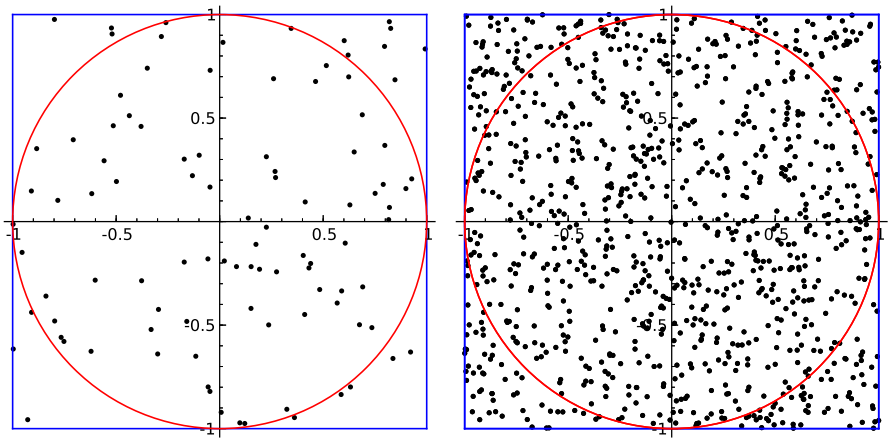


図 36　$[-1,1]^2$ に一様に点をばらまく

上のことをプログラムにすると次のようになります：

```
# モンテカルロ法で円周率を求める
import random
random.seed(0)

def count(n):
    "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"
    p=0     # 以下でカウントする値をpに入れる
    for i in range(n):
        x=random.uniform(-1,1)     # 乱数のx座標
        y=random.uniform(-1,1)     # 乱数のy座標
        if x^2+y^2<1:              # もしx^2+y^2<1ならば
            p=p+1                      # pを一つ増やす
    return p                       # カウントした点の数pを返す

n=100    # サンプル点の個数

pai = 4.0*count(n)/n     # これが円周率
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··· 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。

```
# ここまでは上のプログラム
# の13行目までと同じにする。

def pai(n):
     return 4.0*count(n)/n

for i in range(1,51):
     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
```

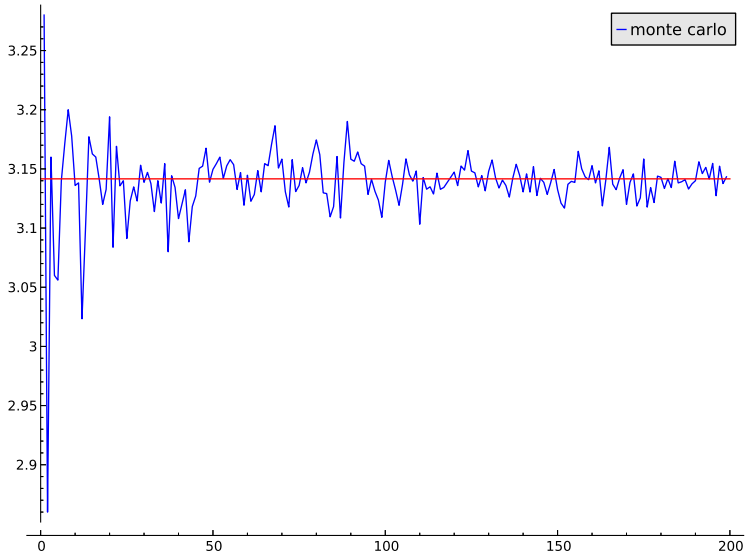
20000 個のサンプル点を取って計算してもそれほ 0.01 の桁が異なっています。これは確率論の基本的な事実で，分散を計算すればすぐにわかります。この方法ではサンプル点の数を n としたときに，円周率の誤差はおよそ $1/\sqrt{n}$ の割合になります。

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：

```
lis = []
for i in range(1,200):
     lis.append([i,pai(100*i)])     # ペア[i,pi(100*i)]のリストを作る
p1 = list_plot(lis, plotjoined=True, legend_label='monte carlo')
```

```
p2 = line([(0,pi),(200,pi)], color='red')     # 円周率の真の値
(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：



50.2.2 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：

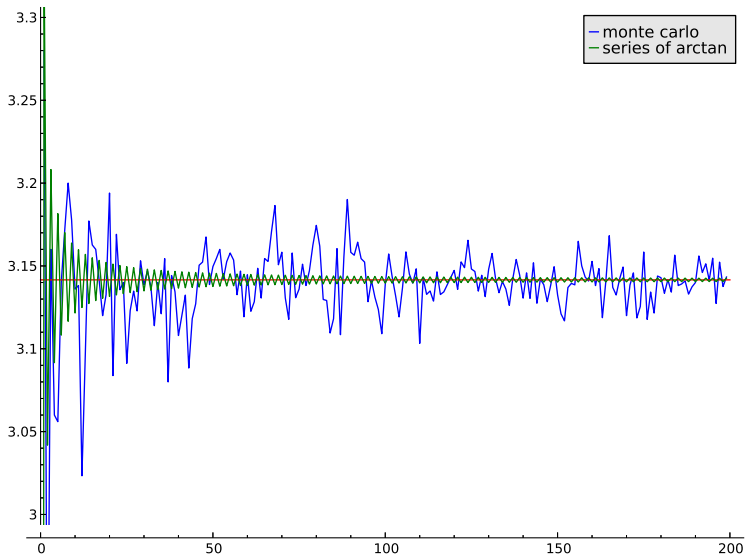
$$\pi = 4\arctan(1) = 4\int_0^1 \frac{1}{1+x^2}dx = 4\int_0^1 \sum_{n=0}^{\infty}(-1)^n x^{2n}dx = 4\sum_{n=0}^{\infty}(-1)^n\frac{1}{2n+1}$$
$$= 4\left(1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots\right)$$

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。

50.3　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

$$B_4 = \{x = (x_1, x_2, x_3, x_4) \in \mathbb{R}^4 | x_1^2 + x_2^2 + x_3^2 + x_4^2 \leq 1\} \tag{34}$$

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

$$|B_4| = \frac{\pi^2}{2} \cong 4.9348 \tag{35}$$

です。計算で得られた結果はこの値に近いでしょうか？

50.4　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

$$\int_0^1 f(x)dx = \frac{1}{N}\sum_{j=1}^{N} f(X_j) + \left(\frac{1}{\sqrt{N}}\text{程度の誤差}\right) \tag{36}$$

が成り立ちます[*27]。したがって，$N \to \infty$ とすれば

$$\int_0^1 f(x)dx = \lim_{N\to\infty}\frac{1}{N}\sum_{j=1}^{N} f(X_j) \tag{37}$$

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

$$\int_0^1 \cdots \int_0^1 f(x_1,\cdots,x_n)dx_1\cdots dx_n \tag{38}$$

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

$$\frac{1}{N}\sum_{j=1}^{N} f(X_j^1,\cdots,X_j^n) \tag{39}$$

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。

50.5　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

$$\int_0^1 dx \int_0^2 dy \int_0^3 dz f(x,y,z) \tag{40}$$

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

[*27] 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。

```
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。