高知大学教育学部の情報数学のテキスト

文責　：　高知大学名誉教授　中村　治

数値計算

コンピュータの活躍している分野は広範囲です。一般の人々に関係する分野でも翻訳ソフトや音声認識や通訳ソフトの性能がよくなっていますし、チェスや将棋や囲碁のソフトや自動車の自動運転に代表される**人工知能**の研究、画像認識や最近将棋や囲碁のソフトが急に強くなったり、インターネットの検索が便利になったり、迷惑メールを判別してくれたり、ネットで買い物をするとおすすめのメールが届くようになる原因になった**機械学習**の研究などがあります。

Python は汎用言語ですがあらゆることに Python が最適の選択である訳ではありません。問題に応じて使うプログラミング言語を選択する必要があります。今は マサチューセッツ工科大学やカリフォルニア大学バークレー校をはじめアメリカの有力大学１０校中８大学（３９大学中２７大学）が Python を初心者の導入教育に使っているみたいですが、数年後には別の言語に入れ替わっている可能性もあります。MIT の教科書は John V. Guttag 著「Introduction to Computation and Programming Using Python」The MIT Press 2013 年 です。youtube (`https://www.youtube.com/watch?v=k6U-i4gXkLM` が第 1 回目） で、「MIT 6.00 Introduction to Computer Science and Programming, Fall 2008」作者：MIT OpenCorseWare として講義が公開されています。現在第 1 回目は延べ 2,561,925 人が視聴しています。第 2 回が 516,316 人、第 3 回が 242,843 人第 4 回が 187,147 人、第 5 回が 187,147 人、第 6 回が 102,661 人、第 7 回が 78,053 人、第 8 回が 84,286 人、第 9 回が 71,735 人、第 10 回が 65,864 人、第１１回が 59,773 人、第 12 回が 58,916 人、第 13 回が 60,340 人、第１４回が 65,568 人、第１５回が 46,156 人、第１６回が 36,041 人、第１７回が 39,321 人、第１８回 34,978 が、第１９回が 31,420 人、第 20 回が 52,812 人、第 21 回が 41,966 人、第 22 回が 32,871 人、第 23 回が 42,762 人そして最終回が 57,041 人です。第 1 回目が概論で一番難しくて、我慢して見ていれば英語にも慣れてきてプログラムを動かしての各論になるので何を話すか予想ができるようになるので後の講義になるほどよく理解できます。二人の教授が登場しますが、最初に登場する教授の英語は聞き取りにくいですが、後から登場するもう一方の教授（John V. Guttag 先生）の英語は聞き取りやすいです。教科書が後から出版されているので少し内容が違います。チョコレートをすぐに食べずに、３０分我慢した子供に幸運が舞い込みます。

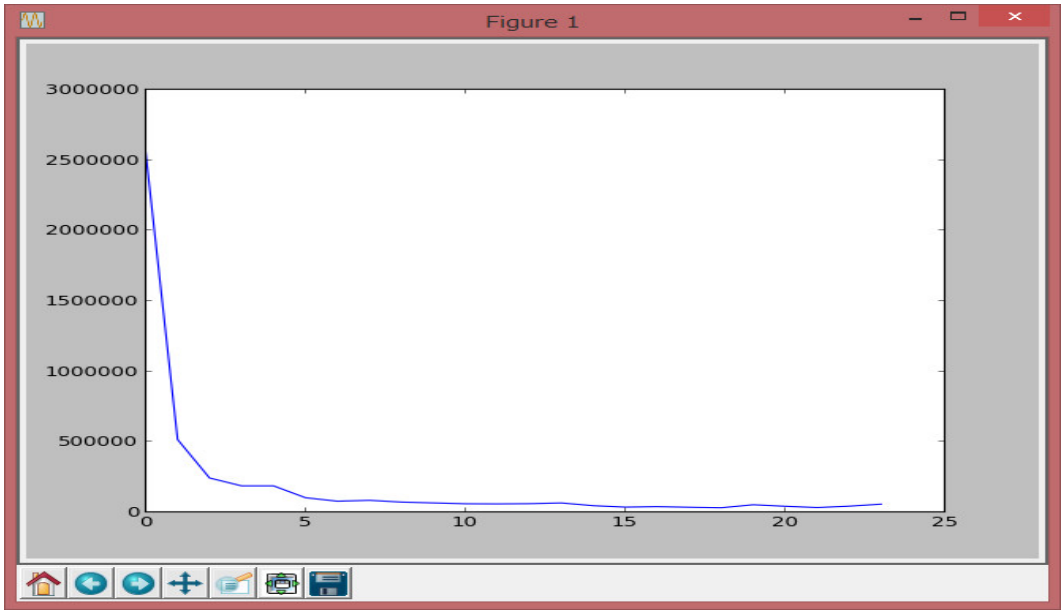
”I don’t regret a thing I’ve done. I only regret the things I didn’t do”(私が後悔するのはしなかったことであり、出来なかったことではない。Ingrid Bergman)

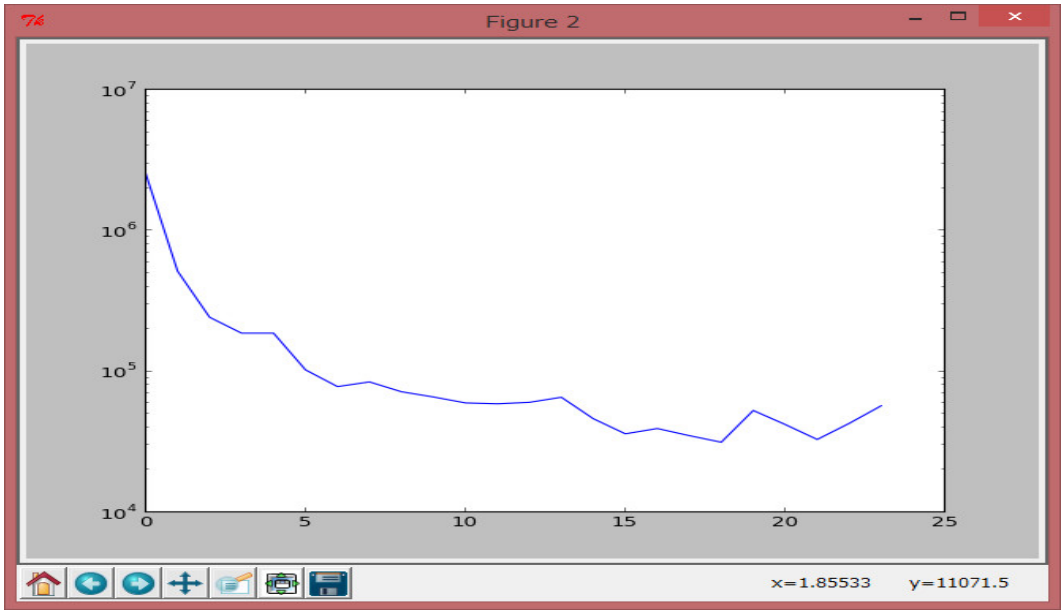
天下の MIT の講義が無料で視聴できます。是非最後まで見てください。良いものを見分けられるためには良いものに沢山触れなければいけません。

```
import pylab
a = [2561925, 516316, 242843, 187147, 187147,
     102661, 78053, 84286, 71735, 65864,
     59773, 58916, 60340, 65568, 46156,
     36041, 39321, 34978, 31420, 52812,
     41966, 32871, 42762, 57041]
pylab.figure(1)
pylab.plot(a)
pylab.figure(2)
```

```
pylab.plot(a)
pylab.semilogy()
pylab.show()
```

を実行すれば、





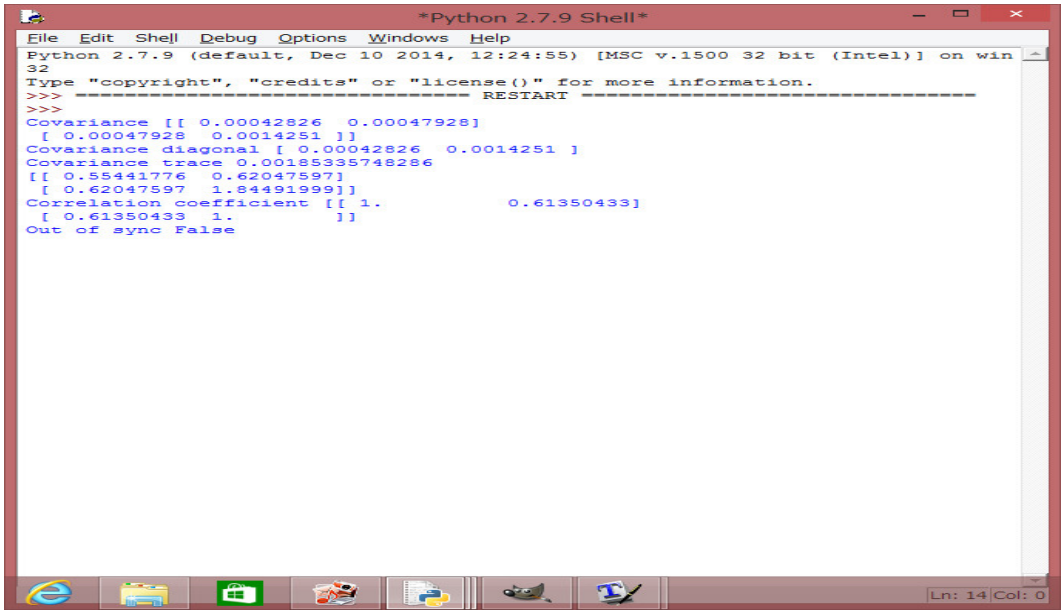
となります。視聴者の変化を示すグラフですがずいぶん印象が違います。下の図は対数目盛を使っています。プログラミングができるようになる人が５０人に１人というデータを裏付けているように思います。you tube には John V. Guttag 氏による「MIT 6.00SC Introduction to Computer Science and Programming, Spring 2011」というのもあります。

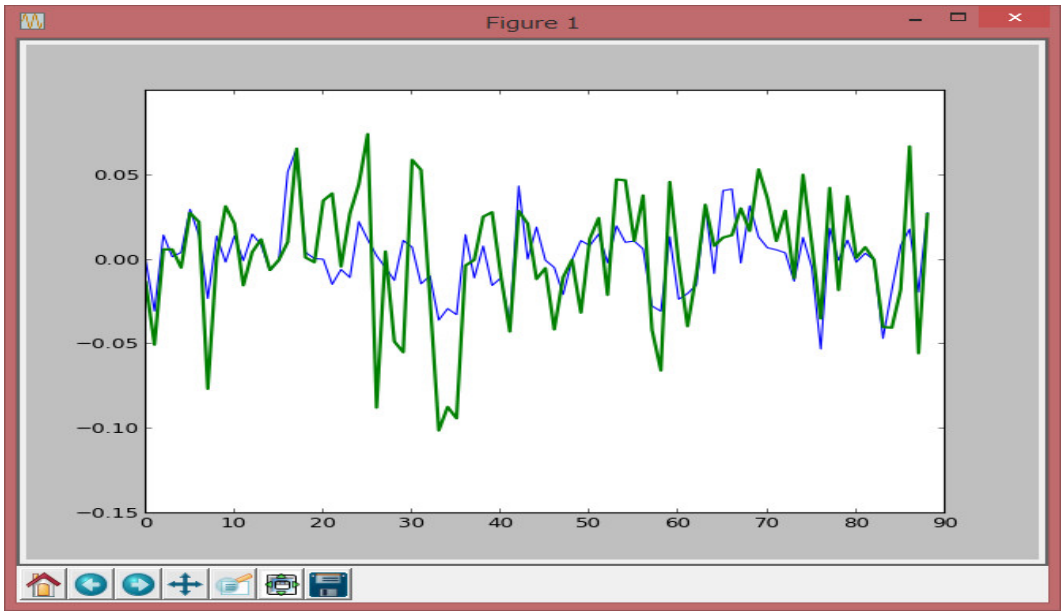
統計学を利用して、大量の情報を視覚化し、株価の予想などもなされています。

Python とそのライブラリー NumPy と matplotlib を使えば

```
import numpy as np
from matplotlib.pyplot import plot
from matplotlib.pyplot import show

bhp = np.loadtxt('BHP.csv', delimiter=',', usecols=(6,), unpack=True)
bhp_returns = np.diff(bhp)/bhp[:-1]
vale = np.loadtxt('VALE.csv', delimiter=',', usecols=(6,), unpack=True)
vale_returns = np.diff(vale)/vale[:-1]

covariance = np.cov(bhp_returns, vale_returns)
print "Covariance", covariance

print "Covariance diagonal", covariance.diagonal()
print "Covariance trace", covariance.trace()

print covariance/(bhp_returns.std()*vale_returns.std())

print "Correlation coefficient", np.corrcoef(bhp_returns, vale_returns)

difference = bhp-vale
avg = np.mean(difference)
dev = np.std(difference)

print "Out of sync", np.abs(difference[-1]-avg) > 2*dev

t = np.arange(len(bhp_returns))
plot(t, bhp_returns, lw=1)
plot(t, vale_returns, lw=2)
show()
```

のようなプログラムを実行すれば、





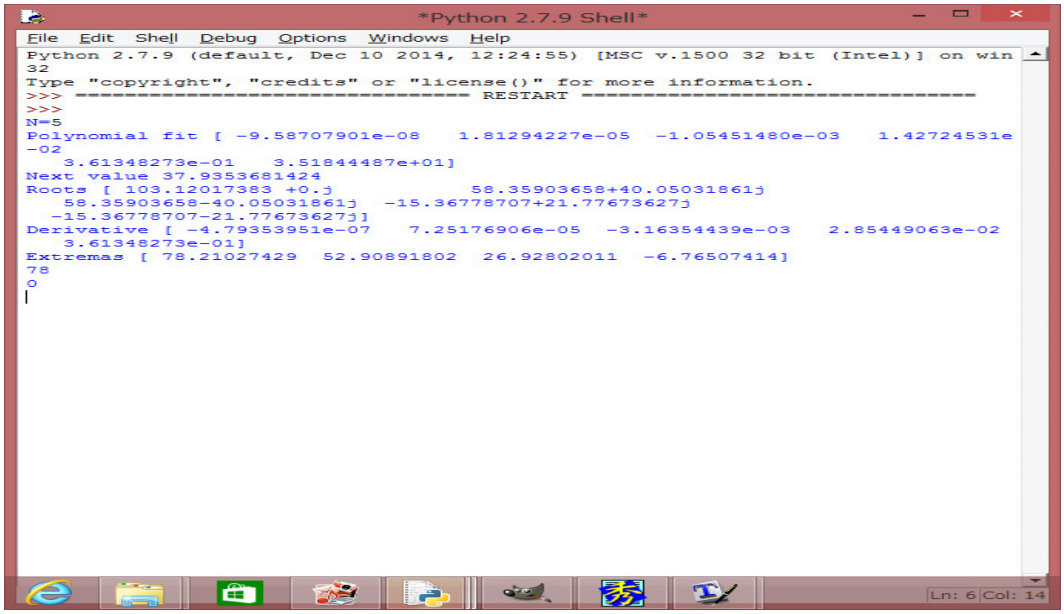
となります。これは、アメリカの金属工業企業の BHP Billiton と Vale の 2015-06-11 から 2015-02-05 までの株価の終値の変動の視覚化とその相関を求めたものです。参考書は Ivan Idris 著「NumPy Beginner's Guide Second Edition」 です。この本のソースコードやデータをダウンロードしようとして、本に書いてある通りしたつもりでもうまくいきません。プログラミングを勉強していると韓国の歴史ドラマのように、次々と災難が降りかかってきます。ドラマでは助けてくれる仲間がいますが、私には助けてくれる仲間はいません。自分で何とかしなければなりません。昔なら途方に暮れるところですが、しかし、今はインターネットがあります。ソースコードは本に書いていますから、それを見て打ち込んでいけばいいので、データの入手が問題になります。著者が BHP Billiton と Vale の株価のデータを使えたということは、どこかに BHP Billiton と Vale の株価のデータがあるはずです。インターネットで探すと BHP Billiton と Vale の 2015-06-11 から 2015-02-05 までの株価のデータがありました。毎日、更新されています。これをゲットし、この本が使っている

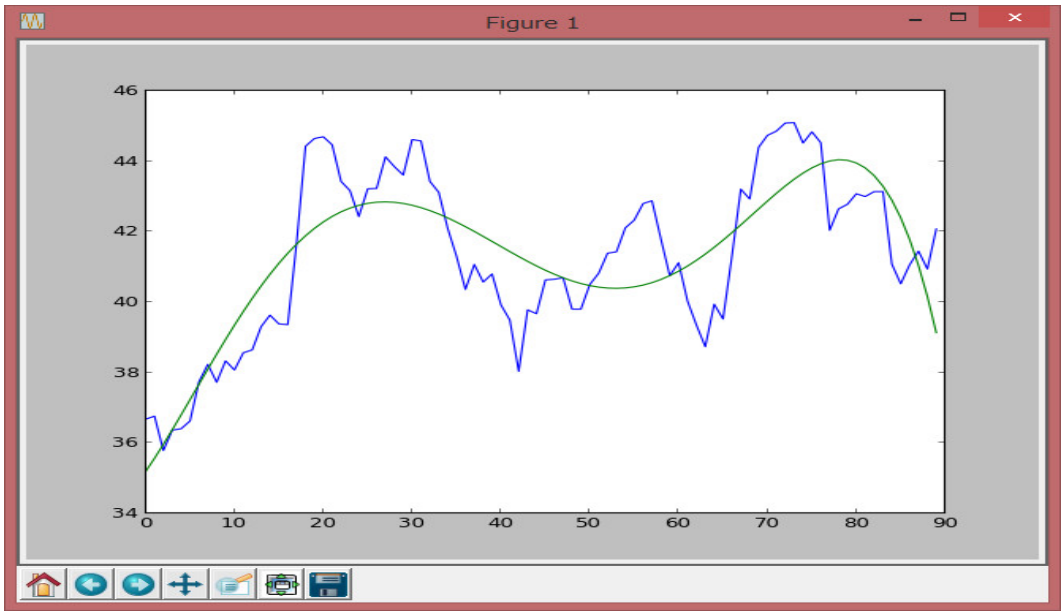
データのフォーマットに作り直すとデータが違うので結果は違いますがともかくこの本のプログラムが実行できます。便利な世の中になりました。

更に、

```
import numpy as np
import sys
from matplotlib.pyplot import plot
from matplotlib.pyplot import show

bhp = np.loadtxt('BHP.csv', delimiter=',', usecols=(6,), unpack=True)
vale = np.loadtxt('VALE.csv', delimiter=',', usecols=(6,), unpack=True)

N = input('N=')
t = np.arange(len(bhp))
poly = np.polyfit(t, bhp-vale, N)
print "Polynomial fit", poly

print "Next value", np.polyval(poly, t[-1]+1)

print "Roots", np.roots(poly)

der = np.polyder(poly)
print "Derivative", der

print "Extremas", np.roots(der)
vals = np.polyval(poly, t)
print np.argmax(vals)
print np.argmin(vals)

plot(t, bhp-vale)
plot(t, vals)
show()
```

のようなプログラムを実行すれば、





となります。これは BHP Billiton と Vale の株価の終値の差を表示し、そのグラフを 5 次多項式で近似しています。最小二乗法の理論を知らなくても、ライブラリー NumPy を使えば簡単に表示できます。どのような曲線で近似すべきかをコンピュータに求めさせることを普通、どの機械学習の教科書でも最初の章で解説しています。Python を使って機械学習を解説している書籍には Willi Richert, Luis Pedro Coelho 著　斎藤康毅訳「実践　機械学習システム」 O'REILLY があり、本をパラパラとめくってみると魅力的な本ですが、これも読むのが大変です。Windows 8 では、この本のプログラムを私は実行することが出来ませんでした。Ubuntu 12 で実行すると本に書いている通りになりません。いくつかの関数が実行できません。仕方がないので最新の Ubuntu 14.04 LTS をノートパソコンにインストールし直して、Python はじめ、必要な最新のライブラリーを別の方法でインストールし直すと本に書いている通り表示するようになりました。ソースコードは本に書いているアドレスからダウンロードできますが、そのソースコードにはデータが揃っていません。

２章で必要なデータも情報が古いのか本に書いているところでは私は見つけることができません。機械学習は大量の囲碁の棋譜から好手と悪手を分類するための法則性を見つけるような研究です。データがなくては話になりません。読むのをあきらめかけていましたが、どっか別のところにあるかもわからないとインターネットで探索して何とか必要なデータを見つけることができました。読み続けるためにはインターネットで探してその都度データを準備しなければなりません。これらのことに気づくまで、試行錯誤を繰り返し、２週間以上時間を浪費しました。英語の原著に対する amazon の書評を見るとこの本のプログラムを実行し、結果を確かめるのがいかに大変かたくさんの人が書き込んでいます。この調子ではいつになったら読み終わるかわかりません。さらに、最近「達人データサイエンティストによる理論と実践　**Python** 機械学習プログラミング」という本が出版されました。この本は読みやすい本で、機械学習の初心者にはお薦めです。「行列プログラマー　**Python** プログラムで学ぶ線形代数」 **Philip N. Klein** 著という非常に面白い本も出版されました。数学として学んだ線形代数とコンピュータで活用するための線形代数とはかなり違うことが分かります。「ゼロから作る **Deep Learning Python** で学ぶディープラーニングの理論と実装」斎藤康毅著というアルファ碁で有名になった深層学習を学ぶための参考書も発売されました。

Python のライブラリーには SciPy という科学計算のためのものもあります。SymPy という数式処理のライブラリーや統計処理のライブラリー Pylab やデータ表示のためのライブラリー matplotlib もあります。Python によって統計学を学ぼうという本もありますが、出版社は情報が古くなると考えているかデータを準備してくれていないので、自分でデータをインターネットで探して入手しなければなりません。統計学も実践的に学ぶためにはデータが不可欠です。Python で OpenCV を学ぼうという本もあります。Pygame というライブラリーを使ってゲームの作り方を紹介した、田中賢一郎著「ゲームを作りながら楽しく学べる Python プログラミング」、「Python ゲームプログラミング　知っておきたい数学と物理の基本」、Al Sweigart 著「Invent Your Own Computer Games with Python with Python」、「Making Games with Python & Pygame」、Will McGugan 著「Beginning Game Development with Python and Pygamw」 などの本があります。「Python クローリング&スクレイピング –データ収集・解析のための実践開発ガイド–」加藤 耕太 著　のような Web 関連の本もあります。現在はこのように **Python** を通して情報科学のいろいろな分野の勉強ができますが、それらすべてに目を通そうとするとあらゆることを犠牲にして勉強しなければなりません。「かくあるべき」ではなく、「かくありたい」と思って勉強しなければ、時間がなんぼあっても足りません。

プログラミングを学び始めたときは次のように考えるといいと私は思います。人間は例によって学び、例によってしか学べません。 興味ある簡単なプログラムをたくさん読んでみます。どうしてこんなことができるんだろうと思うようなプログラムのソースコードを読んでみます。何だ、こんなことをやっているんだ！と思うと思います。不思議なことをやっているプログラムでも、ソースコードを読んでみると人間が手計算でやるように当たり前のことをコツコツやっていることに気づく筈です。人間が手計算でやるように当たり前のことをコツコツやるしかプログラミングできないと心底気づいたとき、プログラムが作れるようになります。また、成功からではなく失敗から人は学びます。上手くいかなくて試行錯誤を繰り返した後、何とかプログラムをでっちあげることができれば、少しプログラミング技術が上達します。人は言わないだけで、これの繰り返しをしています。 「あらゆる失敗を経験したからコンピュータが使えるようになった」という人もいます。いままで Python によるグラフィックスのプログラミングを学んできましたから、すでにいろいろのプログラムを理解できるようになっていますが、数値計算は学んでいません。ライブラリー NumPy や SciPy や SymPy を使えば、計算方法（アルゴリズム）を知らなくても色々面白いことができ

ますし、プログラミングの世界には**「車輪を再発見するな」**という言葉もありますが、皆さんは数学を学び、教職を目指す学生さんです。小学校・中学校・高等学校および大学で学んだ数学が世の中でどのように使われているかを実際のモデル・データに即して実践的に学ぶことも大事ですが、数学の教職を目指す学生さんはそれらの数値計算の基礎になっている数学的概念が何で、どのように計算されているかという基本概念（アルゴリズム）を学ぶことも大切です。パティシエやパティシエールの修業を始める人たちがいきなりお菓子を丸ごと作り始めるのではないように、プログラミングも一足飛びにエキスパートになれるわけではありません。便利なツールを学ぶ前に、地道で面白くないかもわかりません（私が最初に就職した琉球大学理工学部数学科でコンピュータ・プログラミングに初めてふれて、 Fortran で数値計算を学んだ時（琉球大学理工学部数学科の学生さんがコンピュータを学びたいと飲み会で言っていたと私が笠原乾吉先生に伝えると、笠原乾吉先生が手配して、Fortran の集中講義が始まりましたが、わざわざ東京から来ていただいていた集中講義の先生には高度なことを教えて貰おうと助手だった高久章先生が言いだし、「Fortran のイロハ」は助手全員が手分けして学生さんを指導することになり、もっとも助手達は誰もプログラミングを知らないので、実際は計算機が面倒を見るわけですが、学生さんたちが真面目に自習しているか監督しながら、私も学生さんが読んでいる Fortran の自習書を読み、学生さんと一緒にプログラミングの実習をしました。）も数学者ですからやっていることは理解できましたが、代数的位相幾何学（ホモトピー論）が専門で解析学や代数学の専門ではありませんし、その当時（１９７０年代中頃）の計算機を取り巻く状況もありますし（プログラムを書いて計算機室に提出すると一週間後にシンタックスエラーと打ち出された結果が返ってくるので書き直して再度提出するの繰り返し）、また純粋数学の考えが染みついていたので、実社会における近似計算の有用性が理解できていませんでしたので、別に素晴らしいことをやっているとは思いませんでした。しかし、その経験がその後のプログラミング学習 [次に勤務した高知大学では教員には計算機を直接操作させてもらえ、Fortran プログラミングの結果が直ちに分かるようになったし、長い間（十数年）大学の入試結果の処理の為のプログラムを Fortran でプログラミングしましたし、マイコンが発売され、まだ高価（私費で購入し、パソコン本体と白黒ディスプレイ、プリンタ、フロッピーディスクドライブで１００万円しました）でしたが誰でもコンピュータを所有できるようになり、マイコンの BASIC でのパズルやゲームのプログラミングの勉強、今では夢物語ですがバルブ期に教育学部でも購入できたワークステーションを使っての C 言語でのコンピュータグラフィックスの勉強、その後パソコンや大学の共通利用のワークステーション（使えたのはほんの僅かな期間でしたが）の C 言語、後には C++ を使って、ホップ代数のコホモロジーをコンピュータで計算するプログラムの開発、グラフ理論の研究・論文作成のためのプログラムの開発等、主として非数値計算のプログラミングの勉強ばかりでしたが] の基礎になりました。コンピュータの歴史はこのような数値計算から始まっています。多分、一度は通らなければならない道です。）が基本的な数値計算のアルゴリズムとそれをプログラミングする技術を学びましょう。勿論、何を最終目法にしているかによって、何をどのように学ぶべきかは違ってくるとは思いますが。最近、軽視されていますが、一般教養も身に付けるべきです！

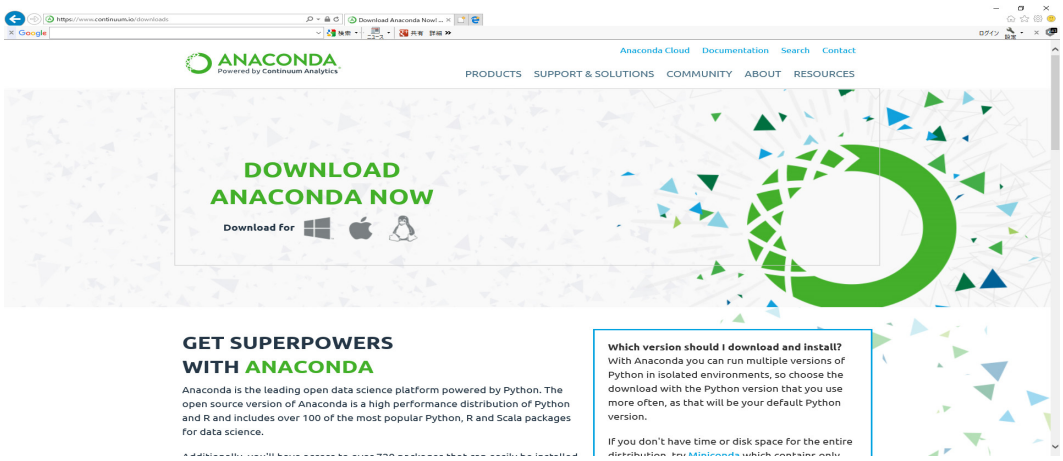
ここでは昔の高等学校の数学 A, B, C に載っていた数値計算のプログラム・アルゴリズム＋α を Python3.6 でプログラミングしてみます。

1　簡単な計算

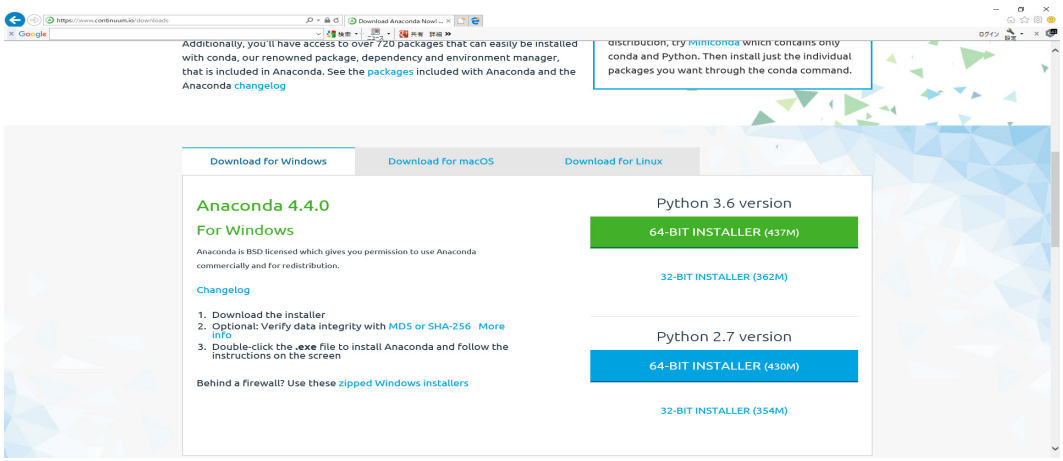
1.1　Python のインストール

まず Python をインストールして、Python を使えるようにしましょう。ここでは Anaconda を使って Python をインストールします。Anaconda は Python 本体に加えて、よく使われるパッケージを一括してインストールできるようにしたものです。

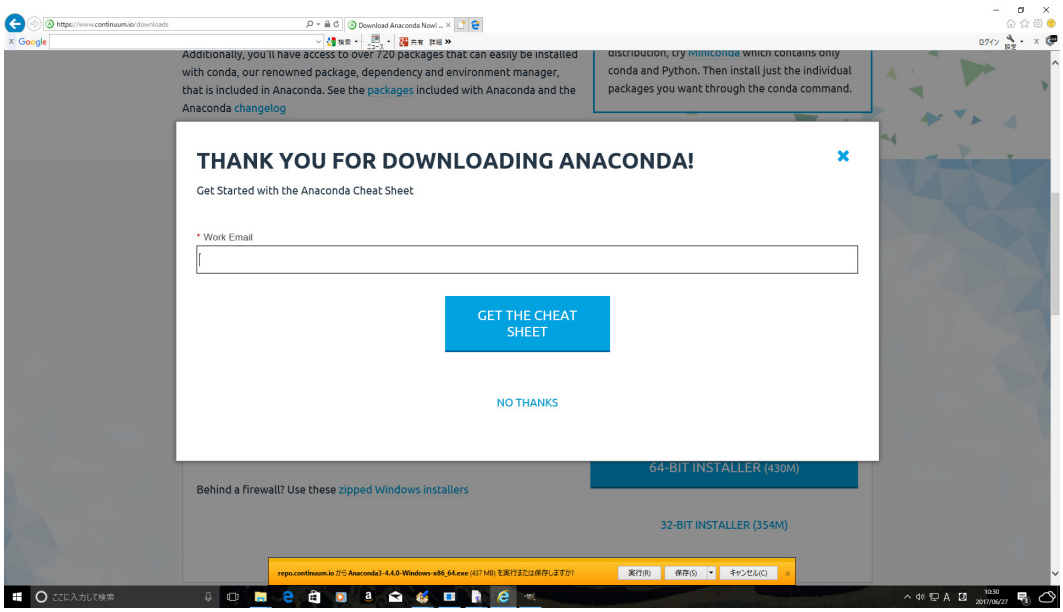
`https://www.continuum.io/downloads`
にアクセスします。



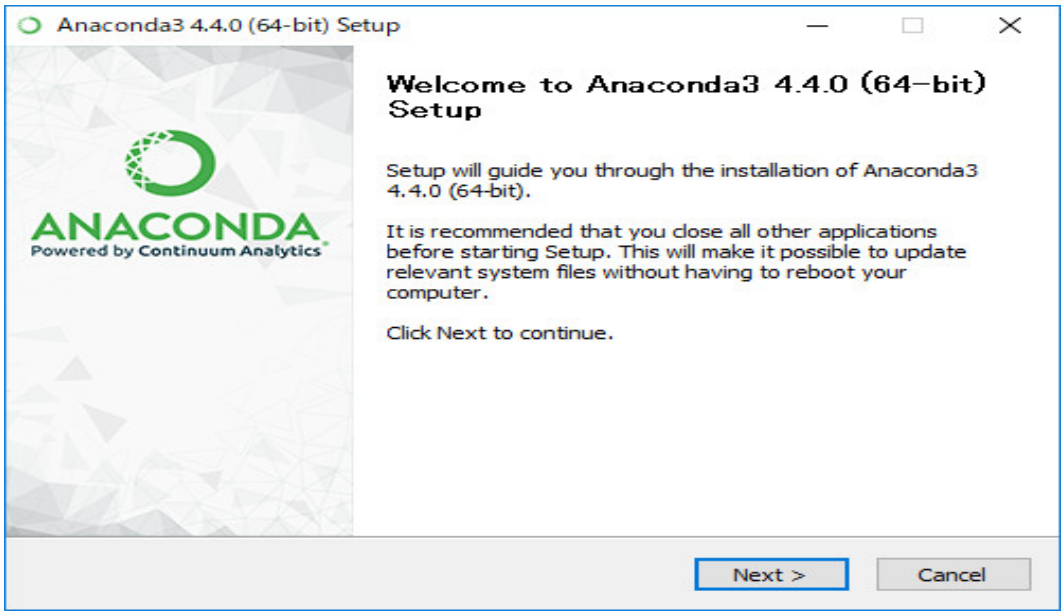
下の方の



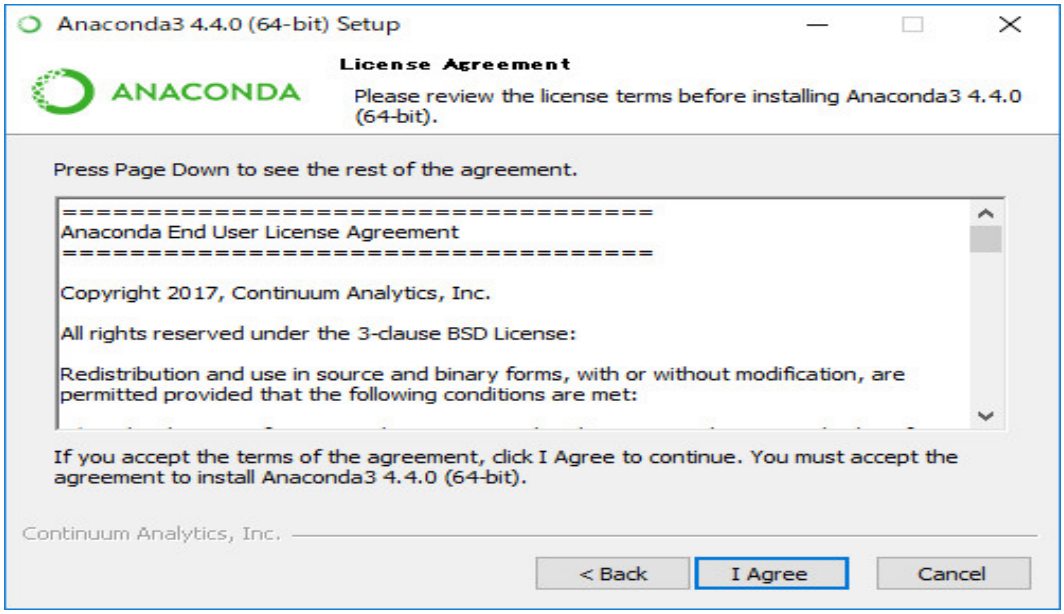
Python 3.6 version をクリックします。現在 Python2 と Python3 が利用できますが、Python 2.7 と Python 3.6 は上位互換性はなく、Python2 は新たな改善はなされないことが決まっているので、これからは Python3 を使って行くのが良いです。



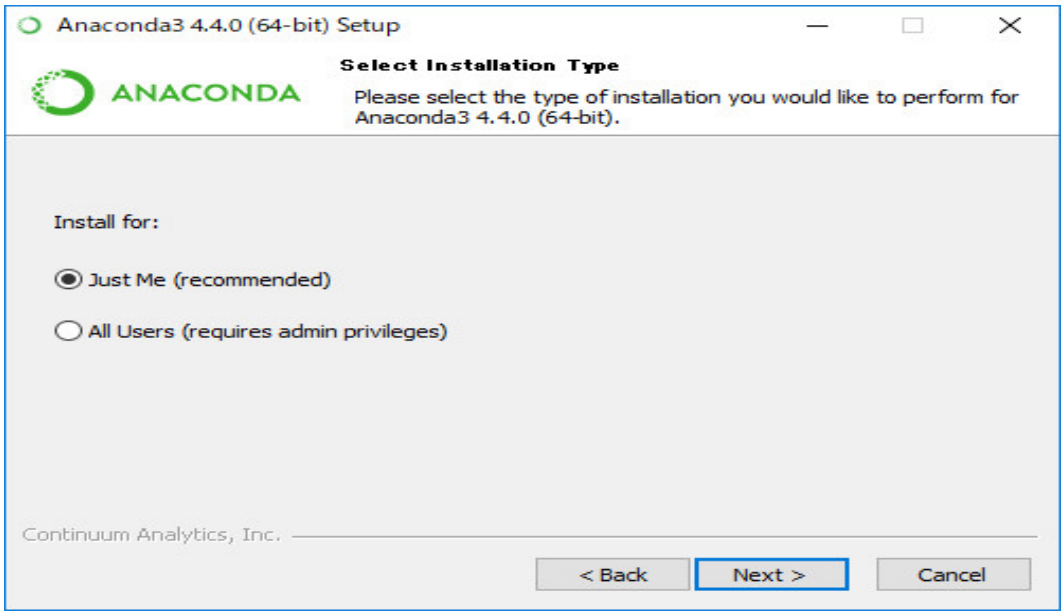
これは無視して、閉じて良いです。「実行」をクリックします。



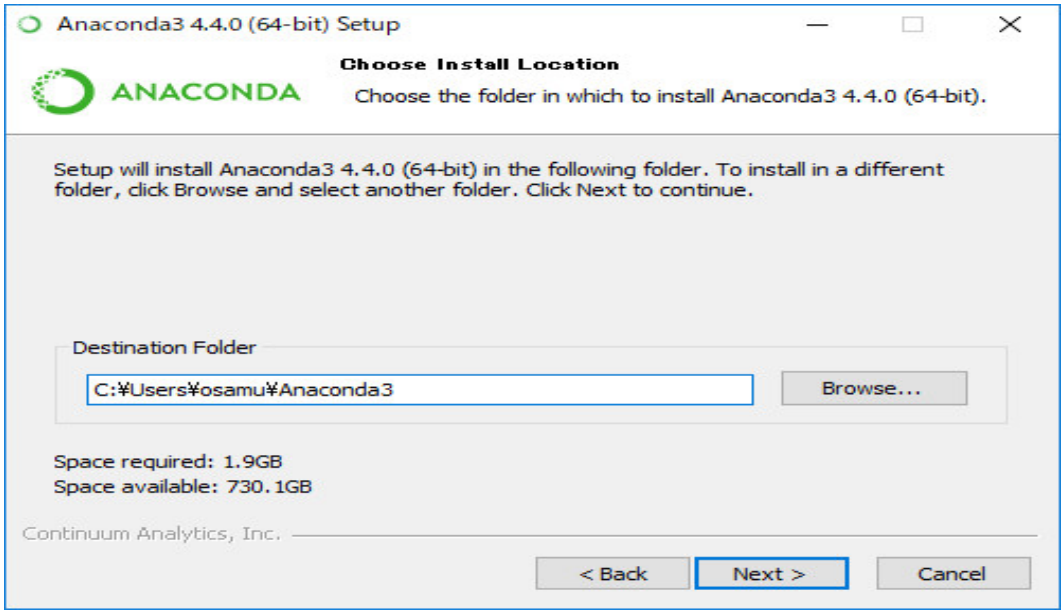
「Next」をクリックします。



「I Agree」をクリックします。



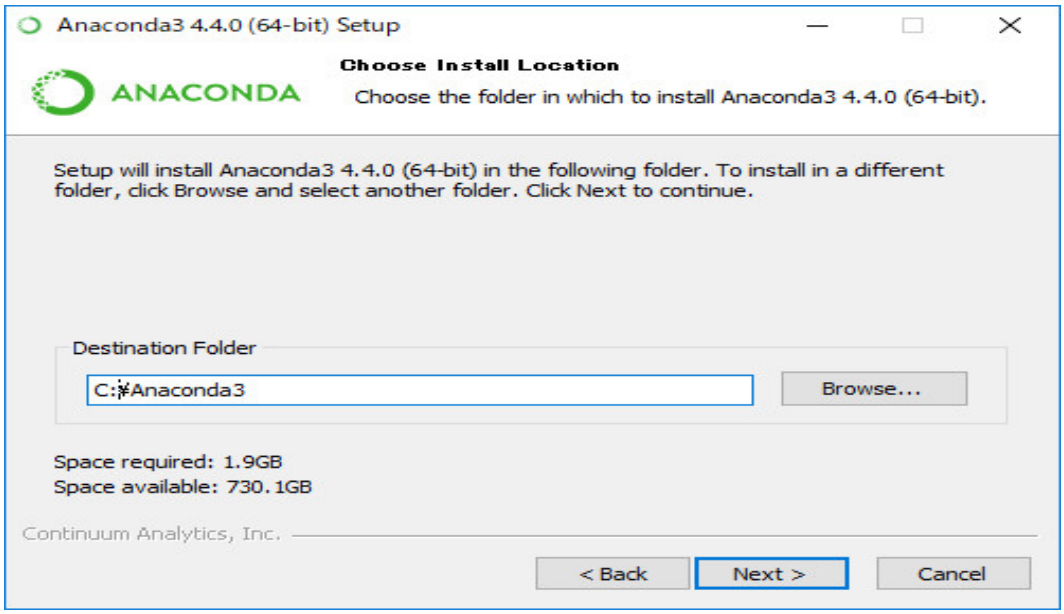
「Next」をクリックします。



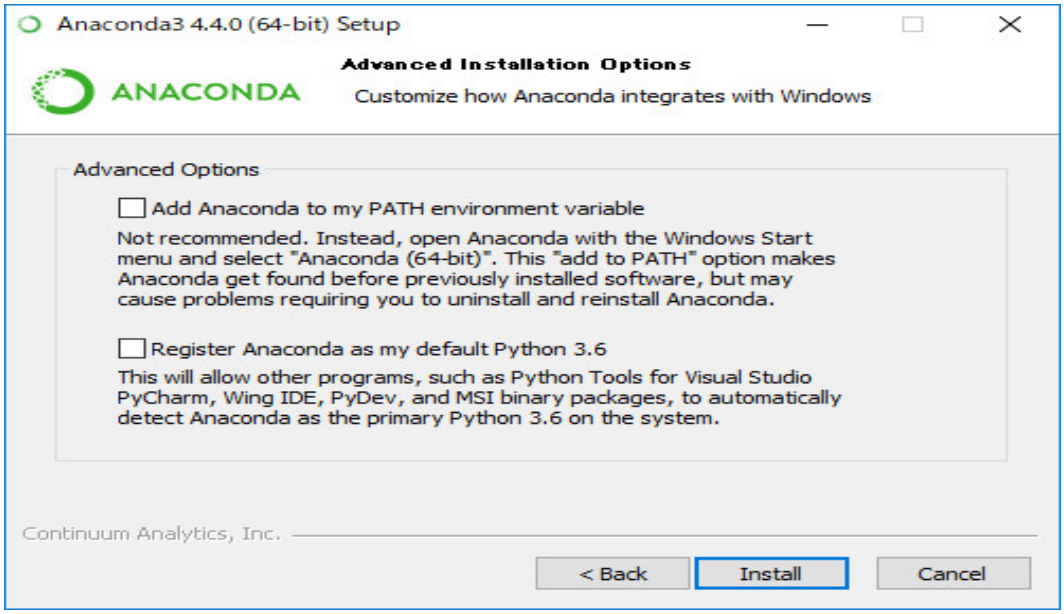
Destination Folder がデフォルトでは、後で色々問題が起こるのでここを

`C:\Anaconda3`

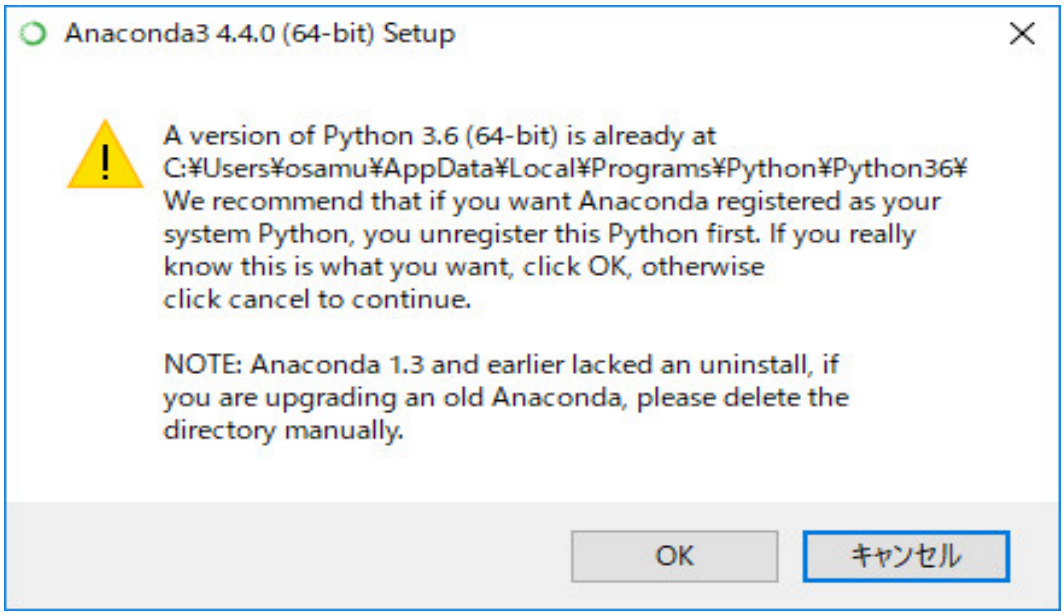
と修正します。



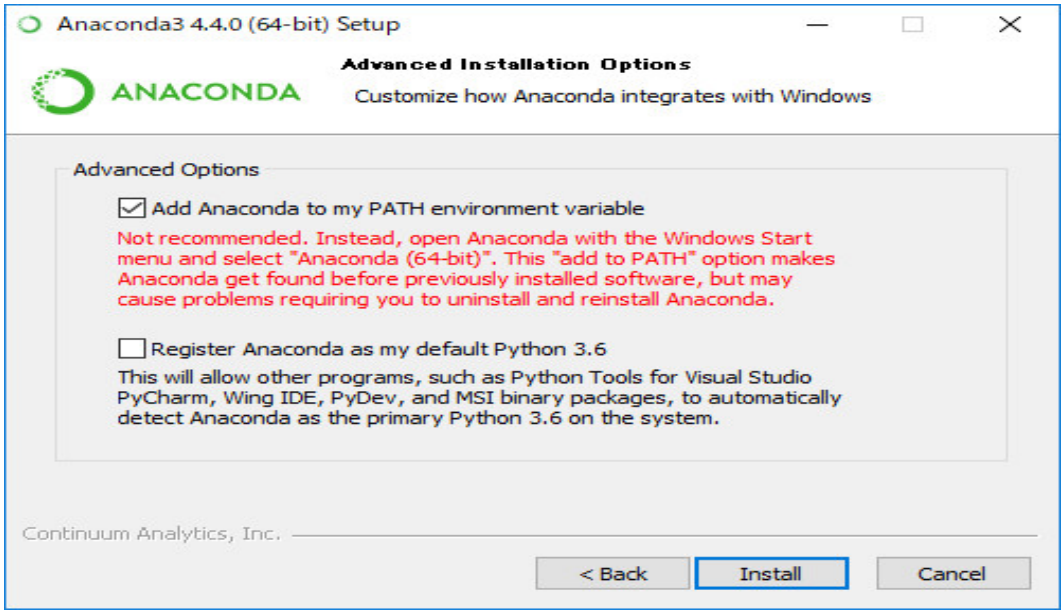
「Next」をクリックします。



通常は両方にチェックを入れます。しかし、すでに Python3.6 をインストールしている場合には、下にチェックを入れると

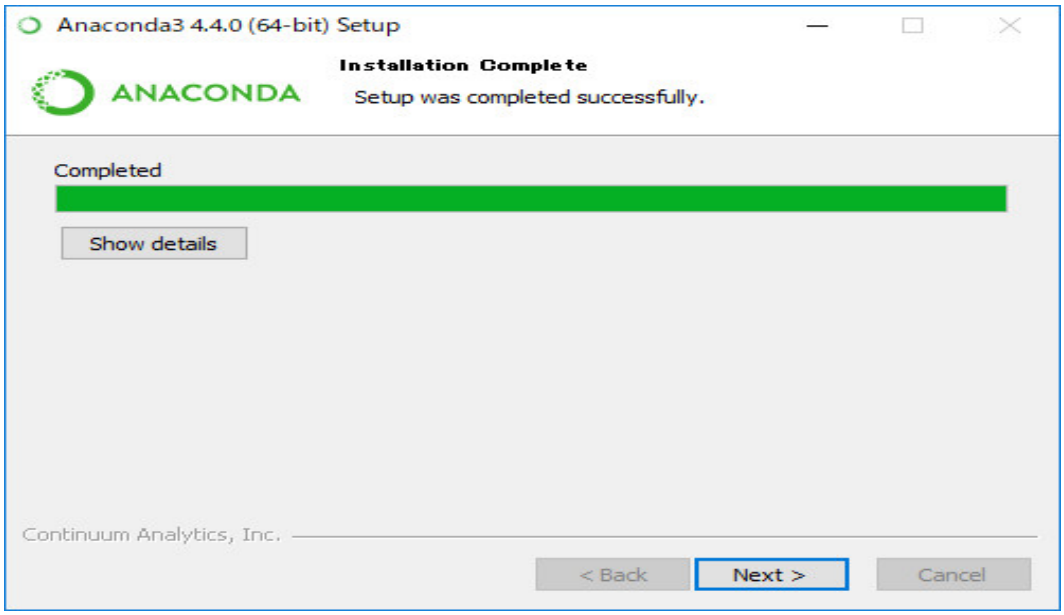


と表示されるので、「キャンセル」をクリックし、すでに Python3.6 をインストールしている場合には

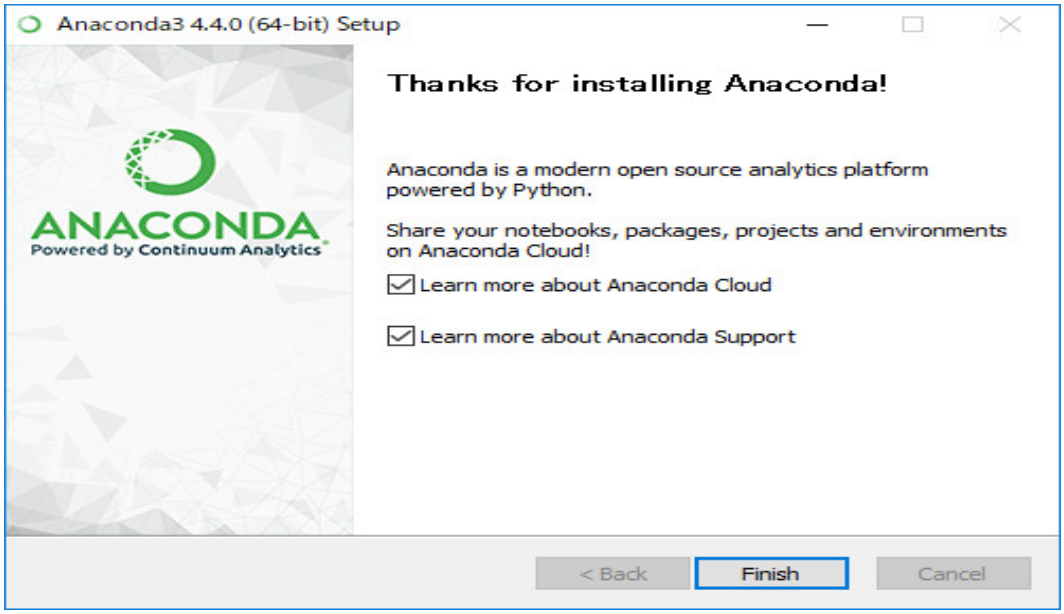


のように、上だけをチェックします。

「Install」をクリックします。

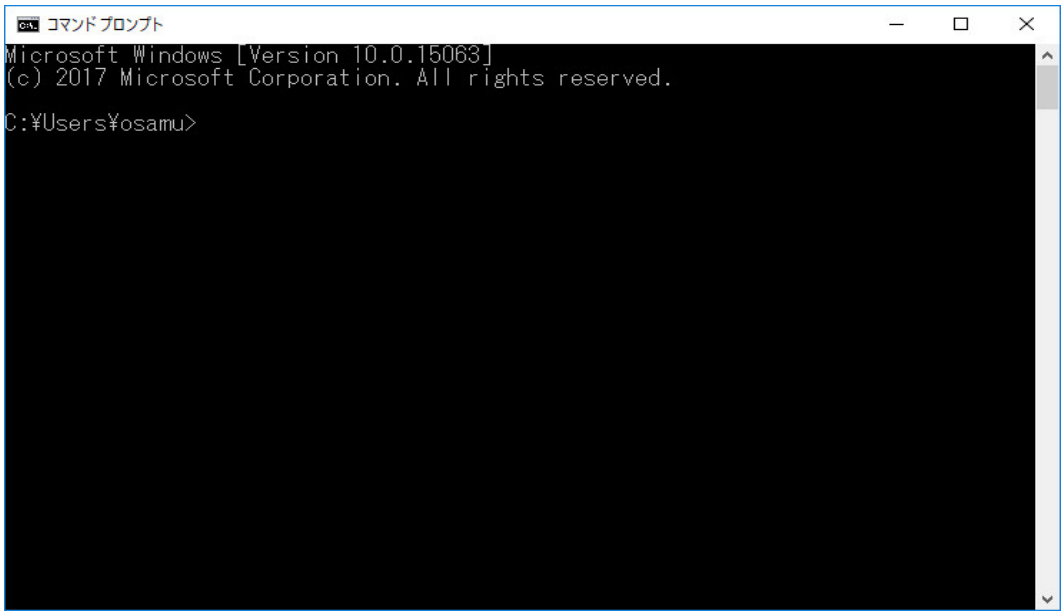


「Next」をクリックします。

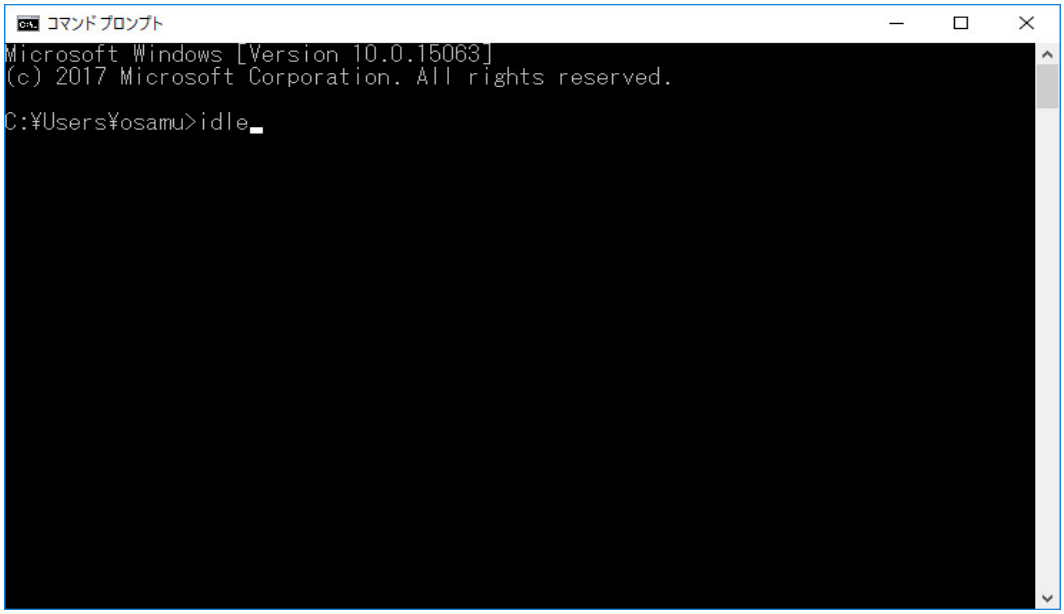


「Finish」をクリックします。

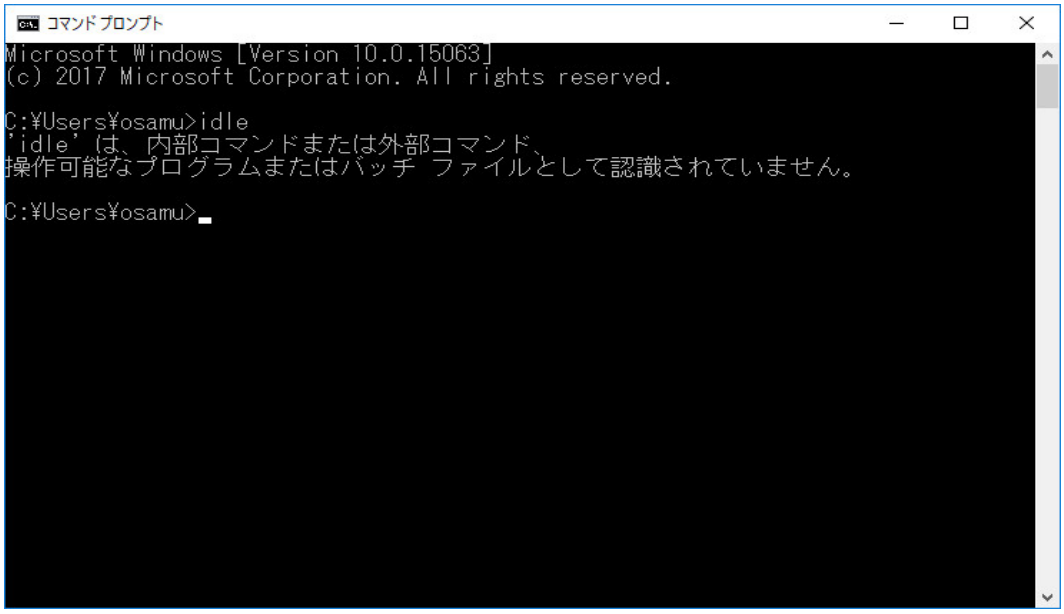
Anaconda をインストールした後に「コマンド　プロンプト」を起動し、



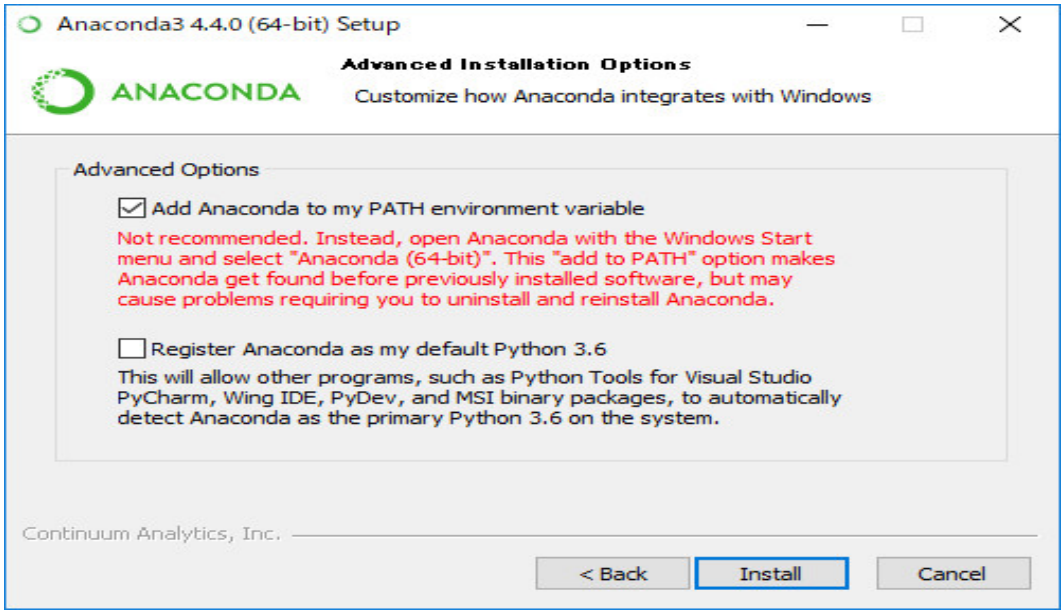
「idle」を



実行します。

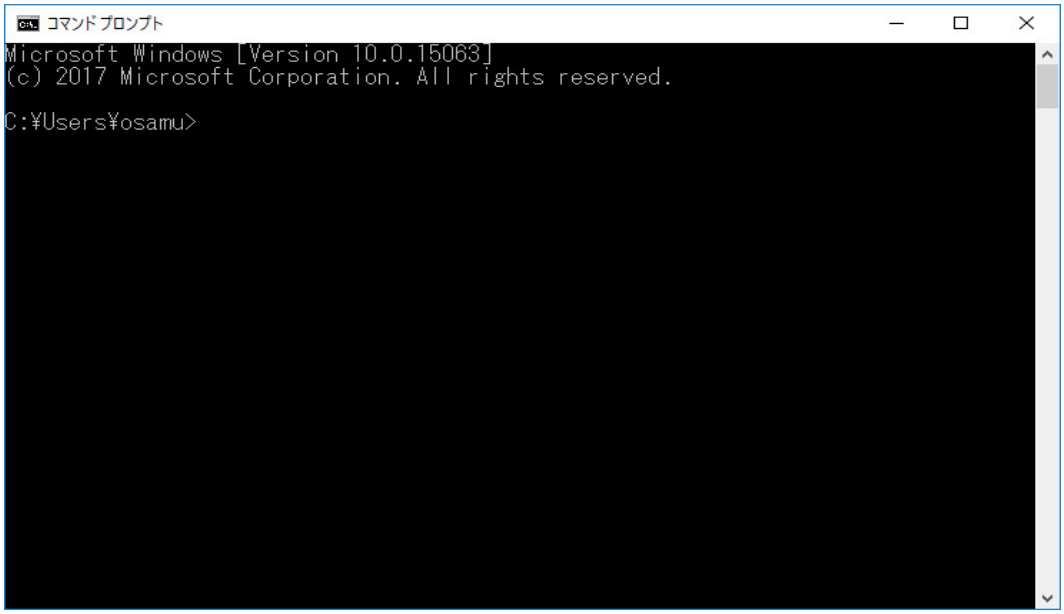


と表示されたら、

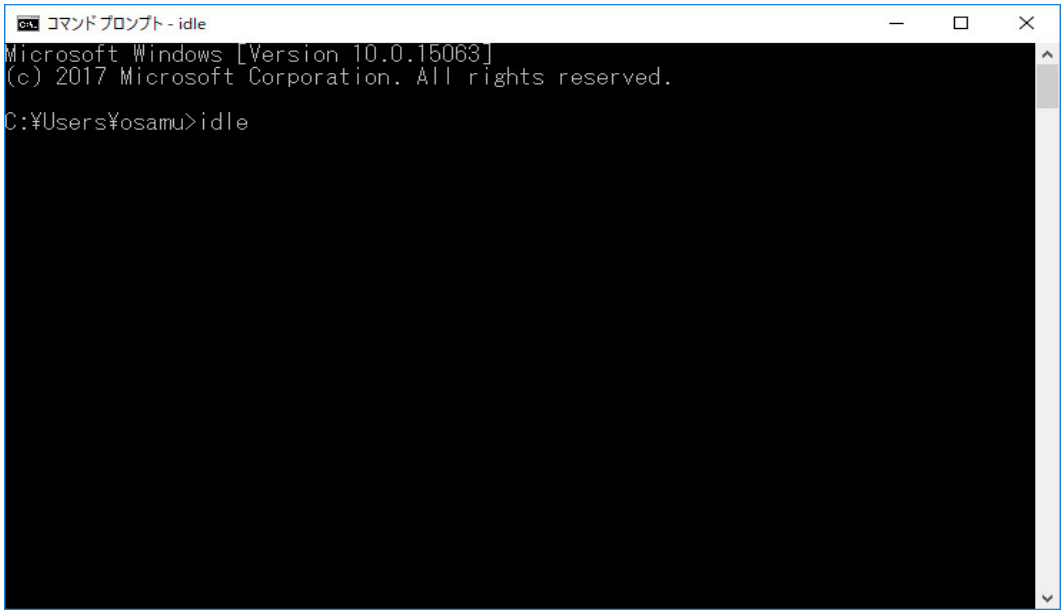


のチェックを忘れていた可能性があります。 Anaconda3 のフォルダーをエクスプローラーで削除して、もう一度インストールをやり直すのが簡単です。

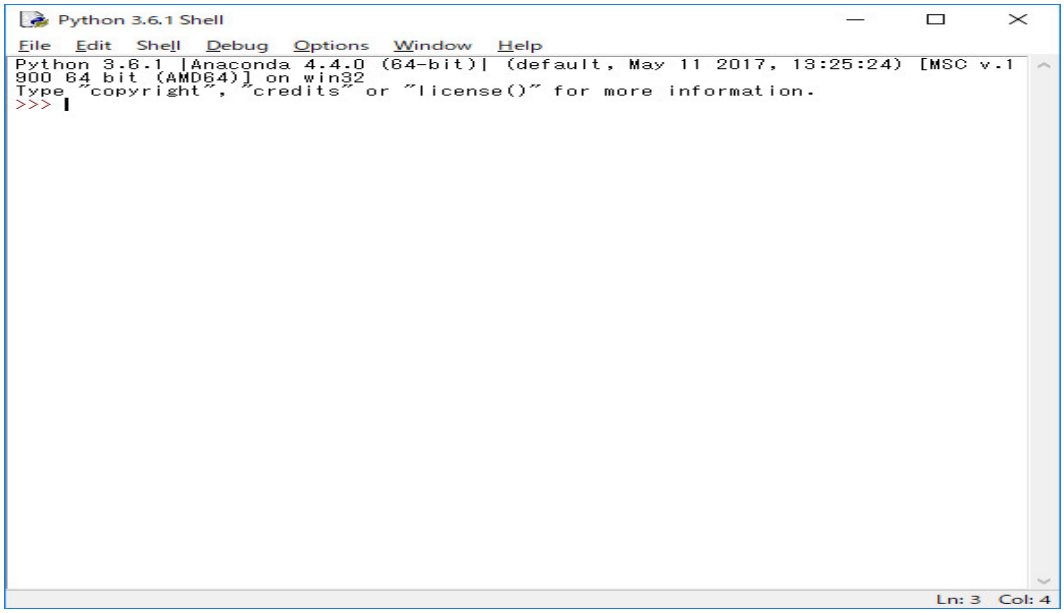
「コマンド　プロンプト」を起動し、



「idle」を

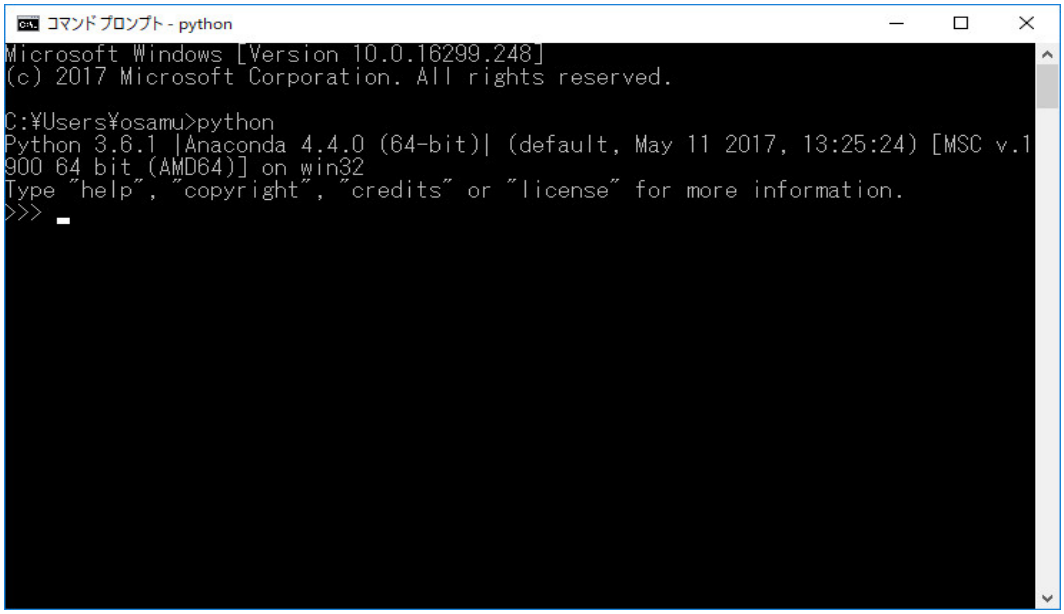


実行します。



と「Python3.6.1 Shell」が起動したら、Anaconda のインストールは完成です。

「コマンド　プロンプト」を起動し、「python」を



実行すると Python インタープリターが走り始めます。

1.2 式と計算

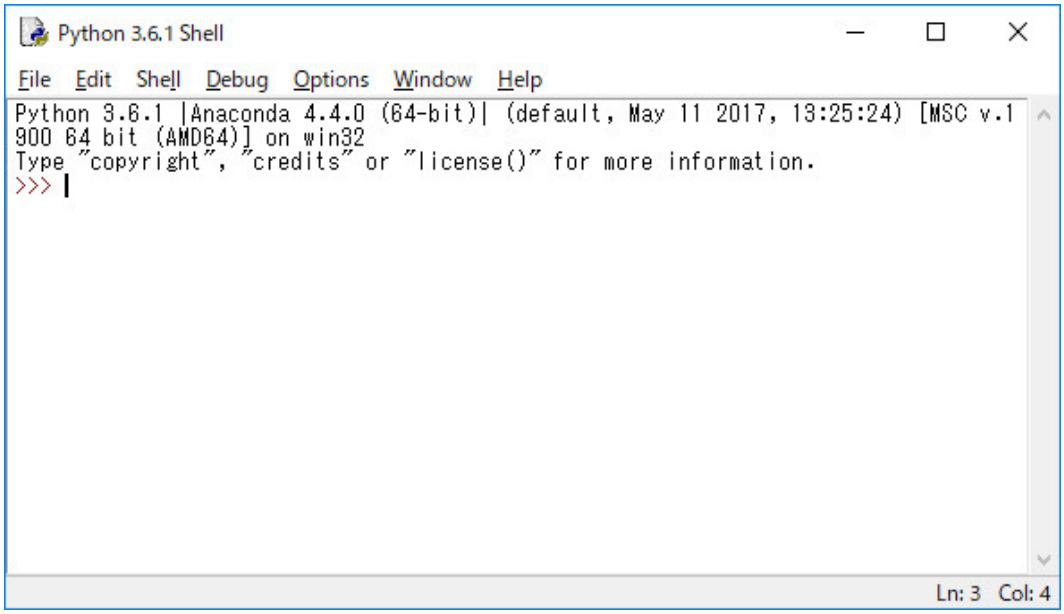
いろいろな計算を行うには、変数と呼ばれるデータの記憶場所に数値を保存させる。変数は名前（変数名）を付けて区別する。変数名はアルファベット、数字、__（アンダーバー）からなる文字列で、先頭の文字には数字は使えない。大文字と小文字は区別するので abc ABC Abc AbC はすべて異なる変数を表す。次の識別子（名前）はキーワード（予約語）として使われるので、別の用途に使えない。

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

それではプログラミングを始めましょう。

例題：三つの数を入力し、その平均と標準偏差を出力するプログラムを作ります。

Python shell (Python GUI) を起動します。



キーボードから数値を入力するには input() 函数を使います。
プロンプト（入力促進記号） >>> に続いて input() と入力します。

```
>>> input()
```

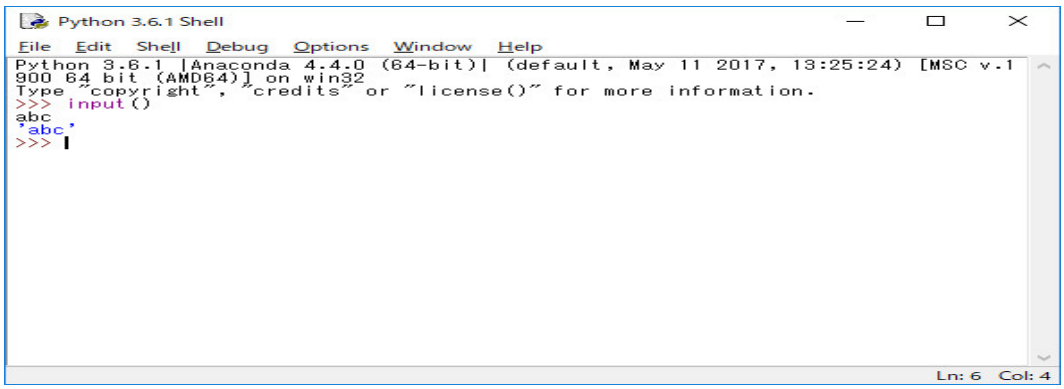
'abc' と打ち込みます。'abc' は abc という文字列を意味します。文字列は "abc" でもいいです。

```
>>> input()
abc
```

'abc' と表示されます。

```
>>> input()
abc
'abc'
```

文字列 'abc' が読み込まれたことが分かります。



しかし、input() だけでは、何を入力すればよいかわかりません。プロンプト（入力促進記号）>>> に続いて input('整数 > ') と入力します。

```
>>> input(’整数 > ’)
```

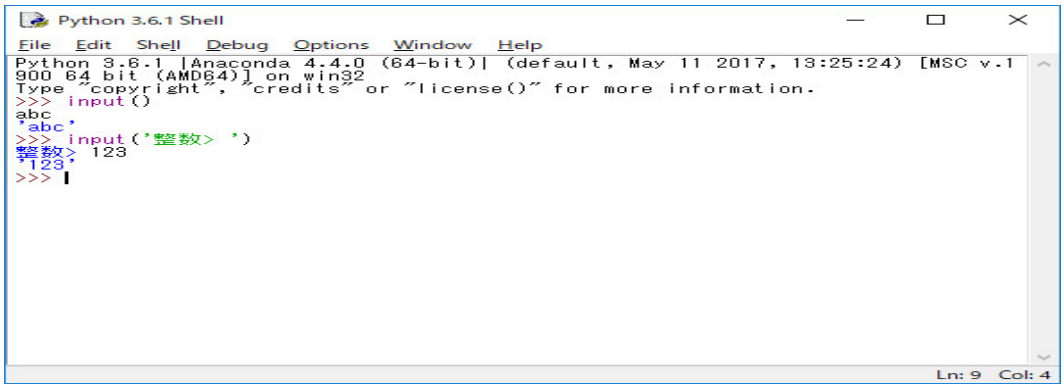
と入力し、
　123 と打ち込みます。

```
>>> input(’整数 > ’)
整数 >123
```

’123’ と表示されます。

```
>>> input(’整数 > ’)
整数 > 123
’123’
```

整数値 123 ではなく、文字列 ’123’ が読み込まれたことが分かります。



　数値を読み込むには、

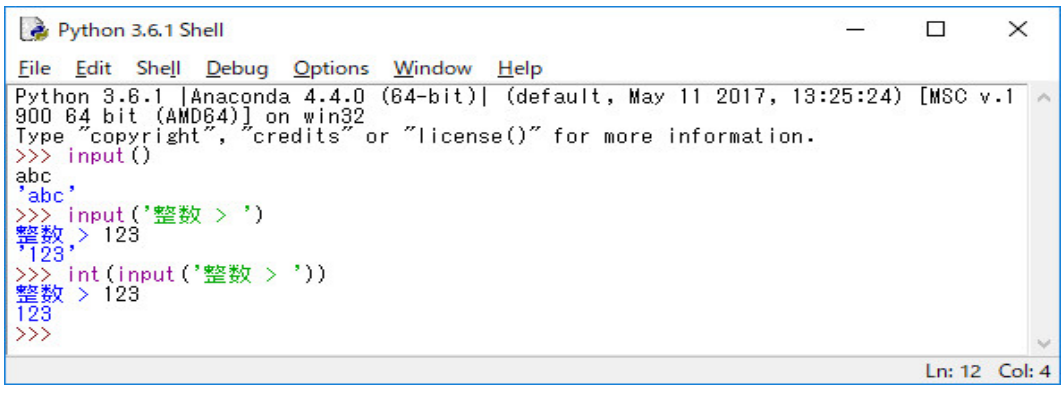
```
>>> int(input(’整数 > ’))
```

と入力し、
　123 と打ち込みます。

```
>>> int(input(’整数 > ’))
整数 >123
```

123 と表示されます。

```
>>> int(input(’整数 > ’))
整数 > 123
123
```



今度は整数値が入力されました。このように整数値を表す文字列を整数値に変換するのに、int() という関数を使います。これも Python 3.6 で変更になった点の 1 つです。Python 2.7 のプログラムを読むときには注意してください。プログラムで使うときは

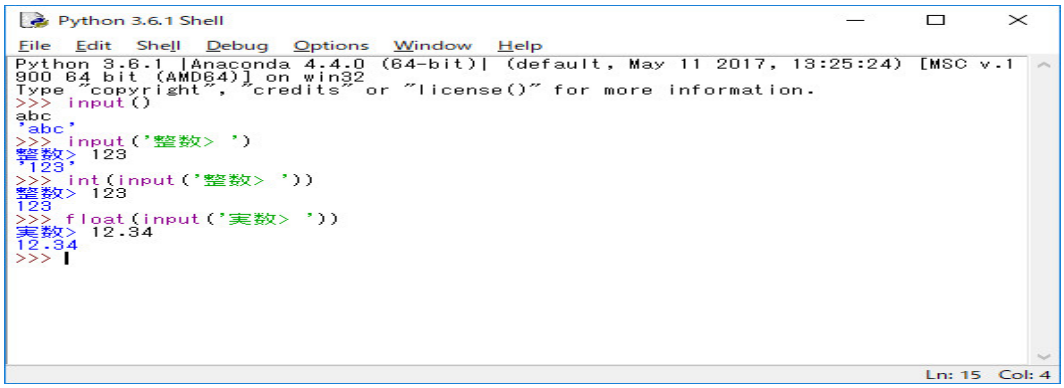
```
>>> n = int(input(' 整数 > '))
```

のように、変数に代入して使います。

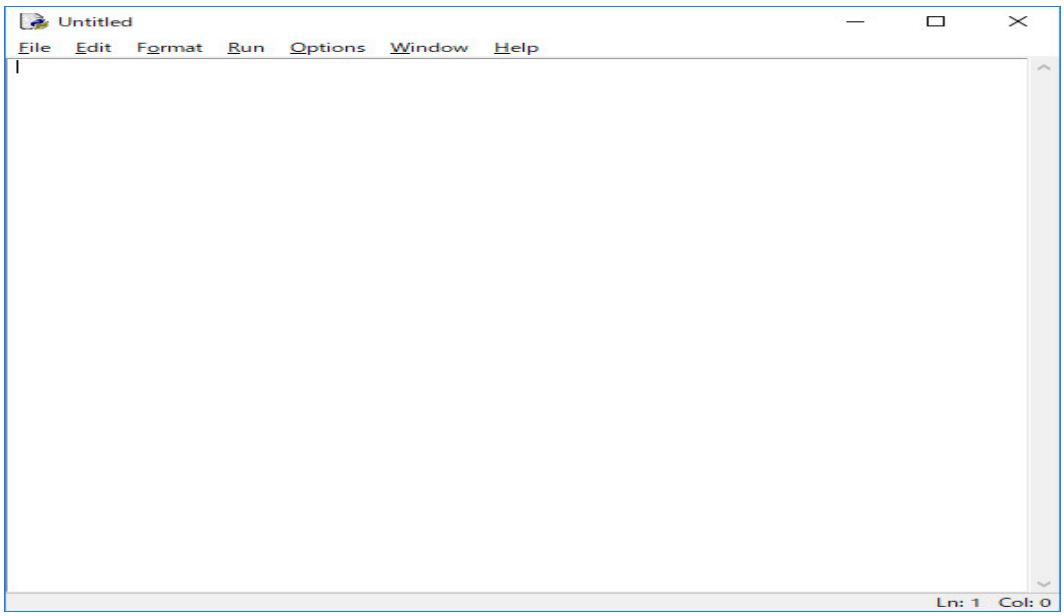
小数点のついた実数や整数値を実数として入力したい時は

```
>>> x = float(input(' 実数 > '))
```

のように、float() という関数で、文字列を実数値に変換します。



File メニューの New File を選択します。

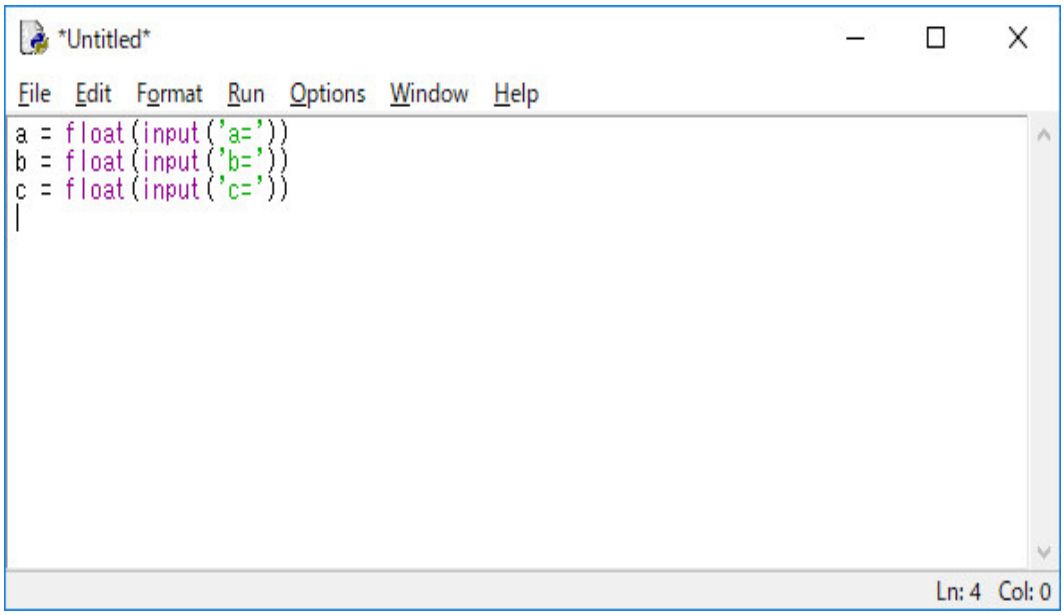


ここにプログラムを打ち込みます。

三つの数を入力してもらうので、その数字を保存する変数が必要です。名前は a と b と c を使うことにします。まず、三つの数字 a, b, c をキーボードから入力してもらうように

```
a = float(input('a='))
b = float(input('b='))
c = float(input('c='))
```

と打ち込みます。



平均と標準偏差を計算するので、それらを保存する変数が必要です。m, s という名前とします。平均と標準偏差を計算する式を追加します。平均は数学の時と同じく式 $(a+b+c)/3$ で表現されます。+ は足し算、/ は割り算を表します。それを m に保存するには

```
m = (a + b + c)/3
```

で表現します。代入文と言います。次は標準偏差の計算です。標準偏差は分散の平方根で与えられます。分散は平均との差の平方の和

$$(a-m)\times(a-m)+(b-m)\times(b-m)+(c-m)\times(c-m)$$

を 3 で割ったものですから

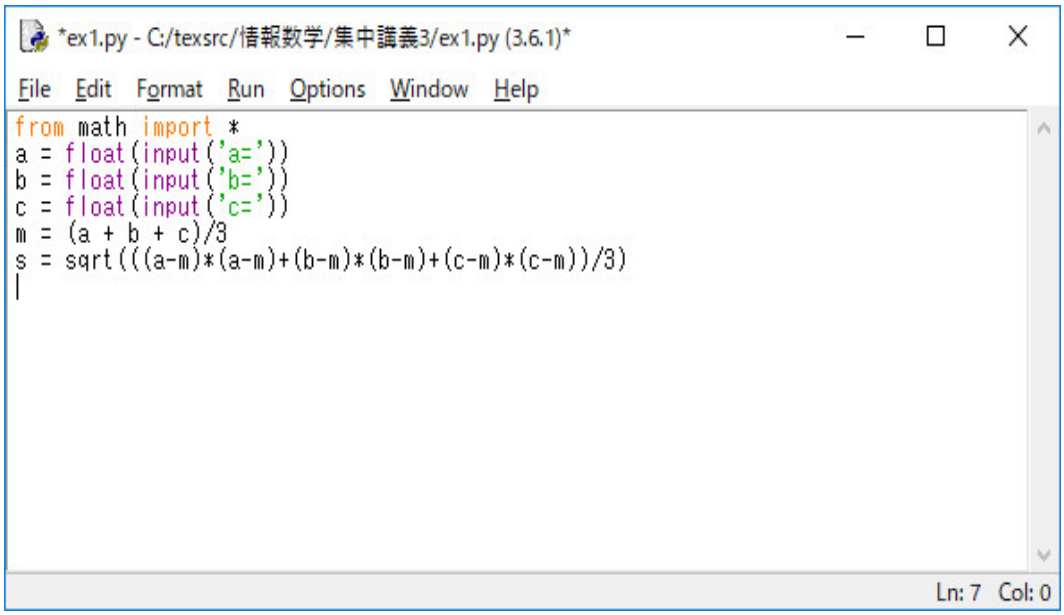
$$((a-m)\times(a-m)+(b-m)\times(b-m)+(c-m)\times(c-m))/3$$

となりますが、Python では × は * で表します。平方根は数学関数 sqrt() で与えられます。これを使うためには math を import する必要があります。

プログラムは

```
from math import *
a = float(input('a='))
b = float(input('b='))
c = float(input('c='))
m = (a + b + c)/3
s = sqrt(((a-m)*(a-m)+(b-m)*(b-m)+(c-m)*(c-m))/3)
```

となりました。



必要な計算が出来たので、結果を表示します。print() という函数を使います。これも Python3.6 の変更点で、Python 2.7 と異なりますので、Python 2.7 のプログラムを読むときは注意してください。Python 2.7 では、print は関数ではなく、文でした。

```
print ("平均　= ",　m)
```

で、「平均　= 」という文字列と m に代入されている実数値が表示され、改行されます。次に標準偏差を表示します。

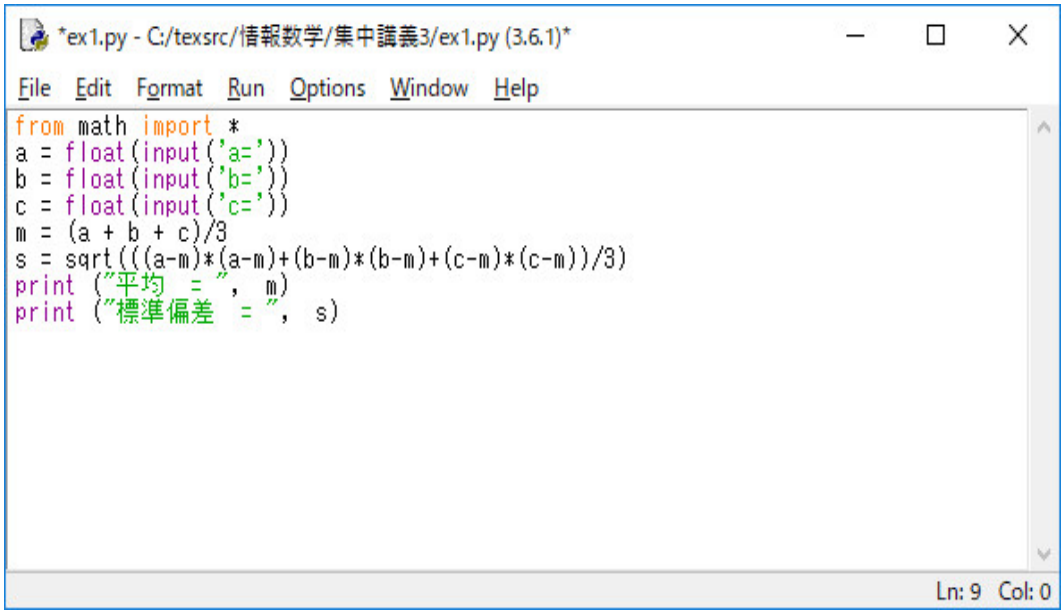
```
print ("標準偏差　= ",　s)
```

で、「標準偏差　= 」という文字列と s に代入されている実数値が表示され、改行されます。

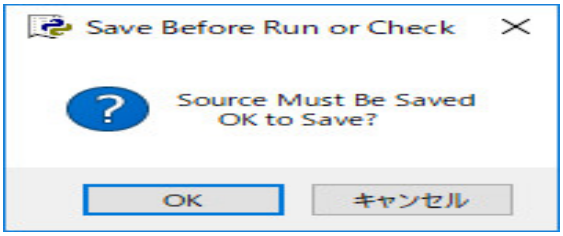
最終的なプログラムは

```
from math import *
a = float(input('a='))
b = float(input('b='))
c = float(input('c='))
m = (a + b + c)/3
s = sqrt(((a-m)*(a-m)+(b-m)*(b-m)+(c-m)*(c-m))/3)
print ("平均　= ",　m)
print ("標準偏差　= ",　s)
```

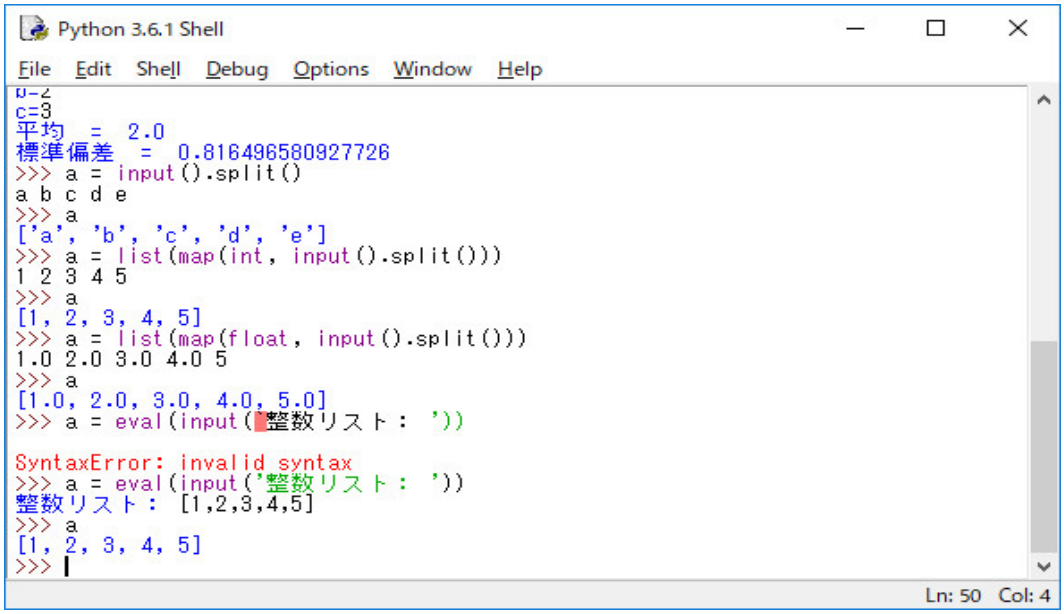
となります。



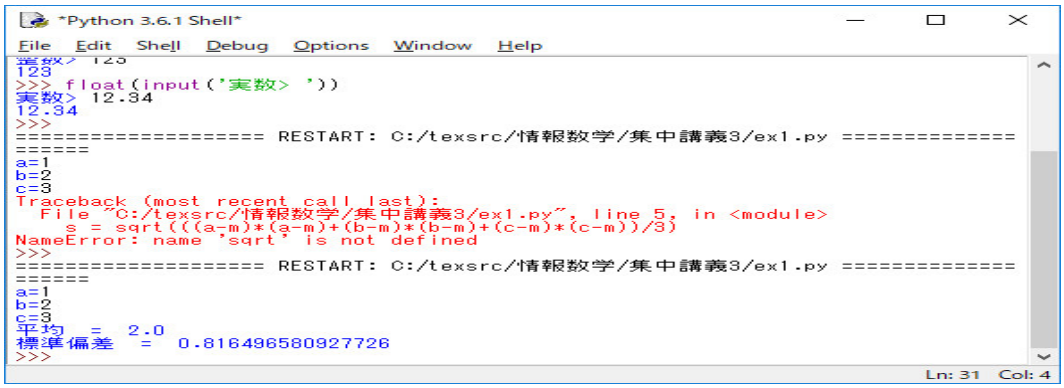
このプログラムを実行します。Run メニューの Run Module を選択し、「Save Before Run or Check」のダイアログの OK のボタンをクリックします。



適当なフォルダに適当な名前（例えば、ex1.py とか hyojyunhensa.py とか) を付けて保存します。Python のプログラムの拡張子は .py です。



Python 3.6 Shell 上でプログラムが実行されますから、a,b,c の数値を入力すると



のように表示されます。エラーメッセージの表示されているブロックは、関数 sqrt() を使っているのに

```
from math import *
```

を忘れていれば、出てきます。間違えたときは、プログラムを修正し、再び実行すれば良いです。

Python 2.7 では、日本語を使ったので警告の Waming ダイアログが出てました。警告が出ないようにしたり、文字化けが起こらないようにするための余計な処理が必要でしたが、Python 3.6 では、その必要がなく、日本語の使用が便利になっています。

Python 2.7 で便利だったのは、input() 函数で、[1,2,3,4,5] のようなリストをキーボード入力すると、[1,2,3,4,5] というリストとして入力出来ていたのが、Python 3.6 では、文字列　a b c d e のキーボード入力なら

```
>>> a = input().split()
```

として、リスト ['a', 'b', 'c', 'd', 'e'] に変換する必要があるし、整数列　1 2 3 4 5 のキーボード入力なら

```
>>> a = list(map(int, input().split()))
```

として、リスト [1, 2, 3, 4, 5] に変換する必要があるし、実数列 1.0 2.0 3.0 4.0 5 のキーボード入力なら

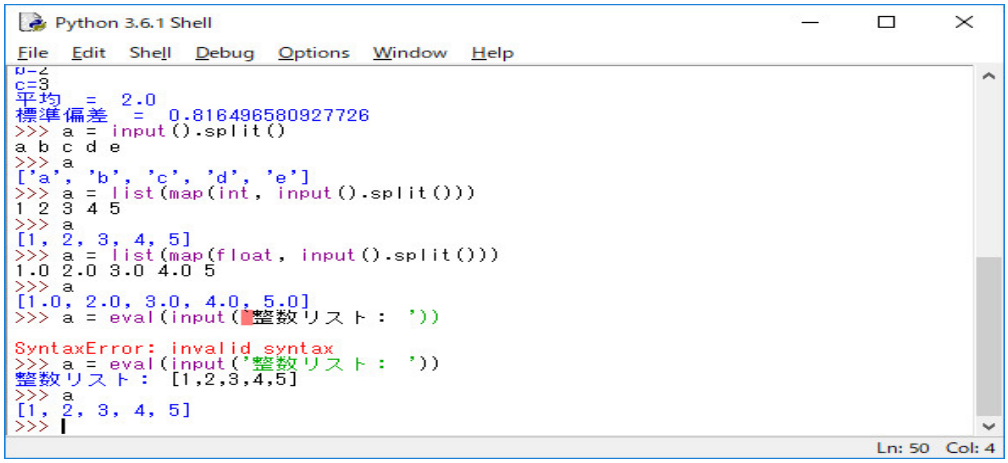
```
>>> a = list(map(float, input().split()))
```

として、リスト [1.0, 2.0, 3.0, 4.0, 5.0] に変換する必要があることです。

別の簡単な方法もあります。[1,2,3,4,5] のキーボード入力で

```
>>> a = eval(input('整数リスト： '))
```

として、リスト [1, 2, 3, 4, 5] を取り込むこともできます。



SyntaxError は、LaTeX の原稿のプログラムをコピーしたために起こりました。何の不具合かわかりませんが、LaTeX の原稿を作るとき、引用符がうまくキーボードから入力できません。このような Python の細かい文法の説明は、この pdf のレベルを超えていますので、キーボードからリストを入力したい時は、いつでもこのようにすれば良いと思って使ってください。

上でみたようにすれば、リストを入力できるので、この問題はリストを使って

```
data = list(map(float, input("numbers list = ").split()))
m = sum(data)/len(data)
v = 0.0
for x in data:
    v += (x-m)**2
s = (v/len(data))**0.5
print (u"平均　= ",　m)
print (u"標準偏差　= ",　s)
```

とプログラミングしてもいいです。

```
m = sum(data)/len(data)
```

の $sum(data)$ は数値のリスト data の各要素の和を計算します。$len(data)$ はリスト data の要素数を計算します。これは for 文を使って

```
m = 0.0
for x in data:
    m += x
m = m / len(data)
```

でもいいです。x には data の各要素が順にセットされます。

```
v = 0.0
for x in datas:
    v += (x-m)**2
```

で、 datas の各要素と平均の差の平方の和を計算しています。$(x-m)**2$ で平方を計算してくれます。

```
    v += (x-m)**2
```

は

```
    v = v + (x-m)**2
```

と同じです。

```
    (x-m)**2
```

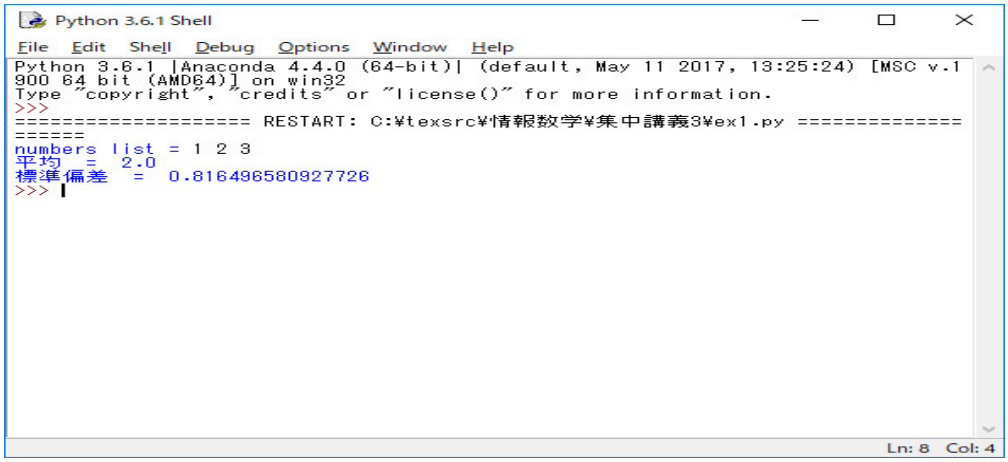
は

```
    (x-m)*(x-m)
```

と同じです。一般に n ** p は n の p 乗です。平方根を sqrt() 函数を使わなくても、

```
s = (v/len(datas))**0.5
```

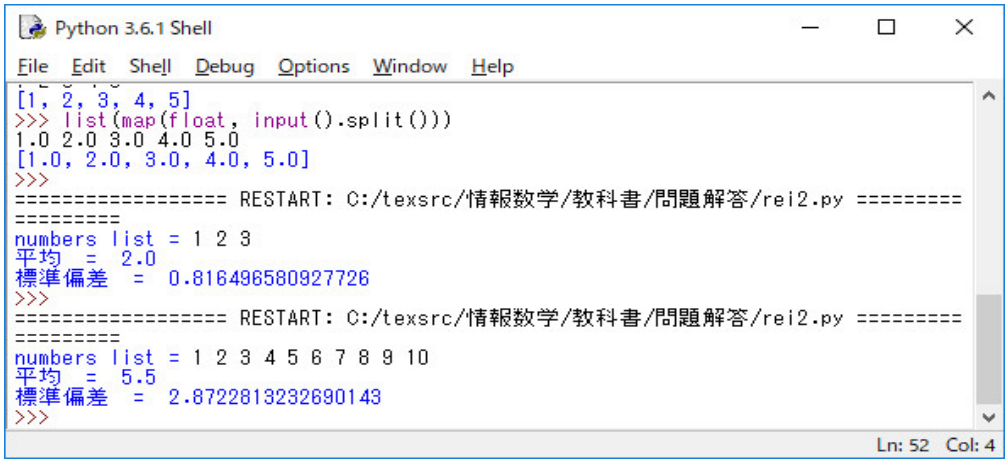
のようにして、 0.5 乗で計算できます。
　実行し、 1 2 3 とキーボード入力すると



です。

```
numbers list = 1 2 3
平均 = 2.0
標準偏差 = 0.816496580927726
```

と表示します。
　再び実行し、1 2 3 4 5 6 7 8 9 10 と入力すると



です。

```
numbers list = 1 2 3 4 5 6 7 8 9 10
平均 = 5.5
標準偏差 = 2.8722813232690143
```

と表示します。

このように、平均、分散、標準偏差の計算式をそのまま Python のプログラムに移すことで、平均、分散、標準偏差の計算をプログラミングできますが、平均、分散、標準偏差の計算式の確認が問題ではなく、計算だけ素早くしたいだけであれば、Python のモジュール（ライブラリー） NumPy を使えば、簡単に計算できます。NumPy を使ったプログラムは

```
import numpy as np
data = list(map(float, input("numbers list = ").split()))
print("平均 = ", np.mean(data))
print("分散 = ", np.var(data))
print("標準偏差 = ", np.std(data))
```

となります。実行すると

```
numbers list = 1 2 3 4 5 6 7 8 9 10
平均 = 5.5
分散 = 8.25
標準偏差 = 2.87228132327
```

と表示します。

```
import numpy as np
```

は numpy をインポートし、以降、numpy を np と省略するという意味です。したがって、

```
import numpy
```

とした時の

```
print  u"平均  = ",  numpy.mean(datas)
print u"分散 = ", numpy.var(datas)
print  u"標準偏差  = ",  numpy.std(datas)
```

ではなく

```
print  u"平均  = ",  np.mean(datas)
print u"分散 = ", np.var(datas)
print  u"標準偏差  = ",  np.std(datas)
```

としています。

```
import numpy as np
data = list(map(float, input("numbers list = ").split()))
print("平均  = ",  np.mean(data))
print("分散 = ", np.var(data))
print("標準偏差  = ",  np.std(data))
```

で、実行できましたが、実際は

```
import numpy as np
l = list(map(float, input("numbers list = ").split()))
data = np.array(l)
print("平均  = ",  np.mean(data))
print("分散 = ", np.var(data))
print("標準偏差  = ",  np.std(data))
```

のように、Python の list を NumPy の array に変換して使うほうが普通です。array には Python の list と異なる算法が定義されています。例えば

```
import numpy as np
import math
lists = list(map(float, input("numbers list = ").split()))
datas = np.array(lists)
mean = float(datas.sum())/ len(datas)
datassqr = datas ** 2
std = math.sqrt((float(datassqr.sum())/len(datassqr))-mean**2)
print("平均  = ",  mean)
print("標準偏差  = ",  std)
```

とプログラミングすることもできます。

```
datassqr = datas ** 2
```

で、datassqr は datas の各要素を平方した array になります。標準偏差は

$$\sigma_X = \sqrt{E(X^2) - E(X)^2}$$

で計算しています。

例題：次に三桁の整数を入力して百、十、一の位の数を表示するプログラムを作ります。

三桁の整数を入力するので、入力された数値を保存する変数を準備します。整数値なので int の変数です。名前は n とします。大した意味は有りませんが、Fortran の伝統で、整数値の変数は、i や j や k や m や n を使うことが多いです。

さてプログラムはまず三桁の整数を入力します。したがって

```
n = int(input('三桁の整数='))
```

となります。

三桁の整数を入力して百、十、一の位の数を表示するので、それぞれのくらいの数字を記憶する変数も必要です。これらを int の変数で、名前をそれぞれ hyaku, juu, iti とします。

三桁の整数の百の位は 100 で割った商です。これは

```
hyaku = n // 100
```

で計算できます。演算子 // は整数を整数で割った時の小数点以下を切り捨てた値を返します。

プログラムは

```
n = int(input('三桁の整数='))
hyaku = n // 100
```

となります。

三桁の整数の十の位を求めるために、もう元の三桁の整数は使いませんから、三桁の整数の百の位を除いた数を n に入れることにします。やり方は複数考えられますがここでは三桁の整数を 100 で割った余りの数を求めることとします。余りを求める演算子は turtle graphics の時出てきた % です。n を 100 で割った余りを n に保存するには

```
n = n % 100
```

とします。＝　は数学の等号と異なり、プログラミングでは代入を意味します。これを追加するとプログラムは

```
n = int(input('三桁の整数='))
hyaku = n // 100
n = n % 100
```

となります。n には二桁（または一桁）の数が代入されているので、十の位を求めるために 10 で割った商ですから、上と同様

```
juu = n // 10
```

で計算できます。これを追加するとプログラムは

```
n = int(input('三桁の整数='))
hyaku = n // 100
n = n % 100
juu = n // 10
```

となります。n の一の位は n を 10 で割った余りですから

```
iti = n % 10
```

で計算できます。これを追加するとプログラムは

```
n = int(input(’三桁の整数=’))
hyaku = n // 100
n = n % 100
juu = n // 10
iti = n % 10
```

となります。次は計算結果の表示です。

```
print("百の位 = ", hyaku)
print("十の位 = ", juu)
print("一の位 = ", iti)
```

で良いです。これを追加するとプログラムは

```
n = int(input(’三桁の整数=’))
hyaku = n // 100
n = n % 100
juu = n // 10
iti = n % 10
print("百の位 = ", hyaku)
print("十の位 = ", juu)
print("一の位 = ", iti)
```

となります。これでプログラムは完成です。
　実はこのプログラムは

```
numstr = input("三桁の整数=")
print("百の位 = ", numstr[0])
print("十の位 = ", numstr[1])
print("一の位 = ", numstr[2])
```

でもいいです。

```
numstr = input("三桁の整数=")
```

で、numstr 入力した三桁の整数が文字列としてセットされます。Python では、文字列もリストのように添え字で書く文字にアクセスできます。ただし、タプルのように値を変更することはできません。
　次のプログラムは atan() 関数を用いて円周率をもとめる Python のプログラムである。

```
import math
print("円周率=", 4 * math.atan(1))
```

　実行すると

円周率= 3.141592653589793

となります。

atan() は浮動小数点数の引数を取り、その arctangent を浮動小数点数として返す関数である。数学関数を用いるためには

```
import math
```

または

```
from math import *
```

のインポート文が必要である。

```
import math
```

とした場合は

```
4 * math.atan(1.0)
```

とし、

```
from math import *
```

とした場合は

```
4 * atan(1.0)
```

とします。

以下にあげるのはよく使われる数学関数である。いずれも一つまたは二つの浮動小数点数の引数をとり、浮動小数点数の値を返す。

sin(x)	x の sine, 単位はラジアン
cos(x)	x の cosine, 単位はラジアン
tan(x)	x の tangent, 単位はラジアン
exp(x)	指数関数 e^x
log(x)	x の自然対数（基数は e）($x > 0$)
log10(x)	x の常用対数（基数は１０）($x > 0$)
pow(x, y)	x^y
sqrt(x)	x の平方根（$x \geq 0$)
fabs(x)	x の絶対値
floor(x)	x を越えない最大の整数
ceil(x)	x 以上の最小の整数
asin(x)	x の arcsine, 返される値の単位はラジアン
acos(x)	x の arccosine, 返される値の単位はラジアン
atan(x)	x の arctangent, 返される値の単位はラジアン

もう、以下の例題のプログラムは一行ずつ解説しなくても理解できるはずです。

例題：球の半径 r の値を入力し、表面積と体積を求める Python のプログラムを作れ。

解答例　これは単に表面積と体積を求める公式を Python の式に変換するだけです。次のようなプログラムを作ればいいです。

```
import math
r = float(input(' 半径 : '))
S = 4.0 * math.pi * r * r
V = 4.0 * math.pi * r * r * r / 3.0
print( "表面積=", S, "体積 =", V)
```

実行すると

```
半径 : 1
表面積= 12.566370614359172 体積 = 4.1887902047863905
```

です。

これは

```
import math
r = float(input(' 半径 : '))
S = 4.0 * math.pi * r ** 2
V = 4.0 * math.pi * r ** 3 / 3.0
print( "表面積=", S, "体積 =", V)
```

でも良いです。

```
S = 4.0 * math.pi * r ** 2
```

や

```
V = 4.0 * math.pi * r ** 3 / 3.0
```

で、演算子 ** の方が演算子 * や / より優先度が高いので、括弧は不要です。括弧を付けても良いです。更に、関数を作って

```
import math
def area(r):
    return 4.0 * math.pi * (r ** 2)
def volume(r):
    return 4.0 * math.pi * (r ** 3) / 3.0
r = float(input(' 半径 : '))
print( "表面積=", area(r), "体積 =", volume(r))
```

とすることもできます。

```
def area(r):
    return 4.0 * math.pi * (r ** 2)
```

は、半径 r の球の表面積 $S = 2\pi r$ を計算し、S を返す関数です。値を返す関数は

```
def <関数名>(<引数の並び>):
    <命令の並び>
    return <返すべき値>
```

の構文で定義されます。 <関数名> は変数名の決め方と同じです。<引数の並び>や <命令の並び> は空でも良いです。

例題：標準体重は 身長 (m)× 身長 (m) ×22 の ±10% 以内です。身長を入力し、その標準体重を出力するプログラムを作れ。

解答例は

```
A = float(input(’身長 (m)=’))
W = A*A*22
L = W * 0.9
H = W * 1.1
print( "標準体重（kg）は", L, "から", H,  "の範囲です")
```

です。

実行すると

```
身長 (m)=1.6
標準体重（kg）は 50.68800000000001 から 61.95200000000001 の範囲です
```

です。

例題： 余弦定理 $c^2 = a^2 + b^2 - 2ab\cos C$ を利用して、a,b,c の値から三角形の角 C を求めるプログラムを作れ。

解答例は

```
import math
a = float(input("a="))
b = float(input("b="))
c = float(input("c="))
C = math.acos((a*a+b*b-c*c)/(2*a*b))/math.pi*180
print( "C=",  C)
```

です。実行すると実行すると

```
a=3
b=4
c=5
C= 90.0
```

です。プログラムをチェックするには、答えの分かっているデータを入力してみます。

2　繰り返しと条件分岐

例題：二つの自然数 N と M を入力し、その最大公約数を出力するプログラムを作れ。

模範解答は「ユークリッドの互除法」を使ったプログラム

```
n = int(input(’n=’))
m = int(input(’m=’))
while m != 0:
    n, m = m, n % m
print( "最大公約数=", n)
```

や

```
def gcd(n, m):
    if m == 0:
        return n
    else:
        return gcd(m, n % m)
n = int(input('n='))
m = int(input('m='))
print( "最大公約数=", gcd(n, m))
```

です。

これらは最大公約数の基本的な性質

```
n > m の時
gcd(n, m) = gcd(n-m, m) = ⋯ = gcd(n % m, m)
gcd(n, 0) = n
```

をそのままプログラミンがしたもので、効率が良いことが知られています。

ここでは練習のために、次のような素朴なアルゴリズムで最大公約数を計算します。プログラミングが出来るようになるためには、模範解答をひたすら覚えるのではなく、思いついたアルゴリズムに基づいてプログラムをいろいろ失敗を繰り返しながら、ひたすら作ってみることです。

今考えるアルゴリズムは

1. N を入力する
2. M を入力する
3. G を N から N-1, N-2, N-3, ⋯, 1 と、G が N と M も共に割り切るまで、一つずつ減らしていく。G が N と M も共に割り切るとき、G は N と M の最大公約数である。
4. G を N と M の最大公約数として表示する。

です。

このアルゴリズムを元にプログラムを作ってみます。

まず、二つの整数を入力します。

```
n = int(input('n='))
m = int(input('m='))
```

です。

g を n から n-1, n-2, n-3, ⋯, 1 と、g が n と m も共に割り切るまで、一つずつ減らしていく。という部分をプログラミングする必要があります。

g を n から n-1, n-2, n-3, ⋯, 1 と、一つずつ減らしていく。というような繰り返しの場合、python では、 for 文か while 文を使いますが、このように繰り返しの規則がはっきりしている場合は普通 for 文を使います。

ここでは練習のために while 文を使ってみます。

今の場合、次の形の while 文を使えば良いです。

while 文は

```
while  条件式:
       命令の並び
```

の形式で、条件式が真 (true) の間はインデントされた命令を繰り返します。今の場合は

```
g = n
while g > 0:
    何らかの処理
    g = g - 1
```

の形式で使います。インデント（字下げ）に注意してください。
一般的な　while 文の基本形は

```
初期設定
while 条件式:
    命令の並び
    後処理
```

の形式で、必要なら初期設定の為の命令を実行し、while 文で、条件式が真であれば、命令の並びを実行し、後処理をし、条件式が真であれば、命令の並びを実行し、後処理をし、を条件式が偽になるまで繰り返します。条件式の一回目の判定が偽であれば、命令の並びは一回も実行されません。後処理は明示的に設定されてなく、命令の並びの中で行われることも有ります。

```
g = n
while g > 0:
    何らかの処理
    g = g - 1
```

の場合は、まず g に n を代入し（g の値を n に保存されている値と同じにする)、g が正であれば (g > 0 は数学の記号と同じ意味です)、 インデントされている何らかの処理を実行し、g をひとつ減らし（g = g - 1)、再び、真ん中の条件式 g > 0 を調べます。g が正であれば、インデントされている何らかの処理を実行し、g をひとつ減らし、を繰り返し、g = n, n-1, n-2, n-3, ⋯, 1 と g の値を変えながらインデントされている何らかの処理を実行します。

```
n = int(input('n='))
m = int(input('m='))
g = n
while g > 0:
    何らかの処理
    g = g - 1
```

となります。
g を n から n-1, n-2, n-3, ⋯, 1 と、g が n と m も共に割り切るまで、一つずつ減らしていく。
の
g が n と m も共に割り切るまで、
の部分を何らかの処理に書きます。条件を判断するには if 文を使います。
if 文は、2 つの形式があります。

```
if 条件式:
    命令の並び
```

の形で、条件式が真 (true) の時、命令の並びを実行するか

```
if 条件式:
    命令の並び
else:
    命令の並び
```

の形で、条件式が真 (true) の時、上の命令の並びを実行し、そうでないとき下の命令の並びを実行します。さらに、if 文はネスト（if 文の中に if 文を埋め込むこと）でき、

```
if 条件式 A:
    命令の並び
else:
    if 条件式 B:
        命令の並び
    else:
        命令の並び
```

を

```
if 条件式 A:
    命令の並び
elif 条件式 B:
    命令の並び
else:
    命令の並び
```

と書くことも出来ます。

```
if 条件式 A:
    命令の並び
elif 条件式 B:
    命令の並び
elif 条件式 C:
    命令の並び
elif 条件式 D:
    命令の並び
else:
    命令の並び
```

と elif を複数並べることも出来ます。

g が n を割り切るは条件式 `n % g == 0` で表現できます。`n % g` は n を g で割った余りを与え、それが 0 と等しいかを判定しています。> や < が大小関係を判定する比較演算子であったように、== は等しいかどうかを判定する比較演算子です。比較演算子は算術演算子より優先順位が低いので括弧は必要ありません。勿論、`(n % g) == 0` と書いても良いです。同様に、g が m を割り切る

は条件式 m % g == 0 で表現できます。これが同時に成り立つことを問題にしているので論理演算子「かつ」で連結すればいいです。「かつ」を表す論理演算子は and で表現します。論理演算子は比較演算子より優先順位が低いので括弧は必要ありません。従って、n % g == 0 and m % g == 0 で g が n と m も共に割り切るかどうか判定できます。勿論、(n % g == 0) and (m % g == 0) としても良いです。

従って、今の場合、if 文は

```
    if  n % g == 0 and m % g == 0:
```

となります。

g を n から n-1, n-2, n-3, ···, 1 と、g が n と m も共に割り切るまで、一つずつ減らしていく。ですから、条件式 n % g == 0 and m % g == 0 が成り立てば、g を減らすのをやめますから、while 文から抜け出す必要があります。この為の命令が break 文です。

従って if 文は

```
    if  n % g == 0 and m % g == 0:
        break
```

となります。これをプログラムに追加します。

```
n = int(input('n='))
m = int(input('m='))
g = n
while g > 0:
    if  n % g == 0 and m % g == 0:
        break
    g = g - 1
```

となります。 break 文を実行したとき、g には n と m の最大公約数がセットされていますから。アルゴリズムの最後の行

g を n と m の最大公約数として表示する。

を実行します。

```
print( "最大公約数=", g)
```

を追加します。

```
n = int(input('n='))
m = int(input('m='))
g = n
while g > 0:
    if  n % g == 0 and m % g == 0:
        break
    g = g - 1
print( "最大公約数=", g)
```

となります。プログラムが完成しました。

実行してみます。

```
n=120
m=96
最大公約数= 24
```

となります。

昔、小学校で習った最大公約数の求め方をプログラミングしてみましょう。今度のアルゴリズムは

1. N を入力する
2. M を入力する
3. D = 1 とする
4. 次の処理を繰り返す
   (a) M と N を共に割り切る G を求める。G=1 しかなければこのステップを終わる
   (b) N = N / G とする
   (c) M = M / G とする
   (d) D = D * G とする
5. D を N と M の最大公約数として表示する。

です。ステップ 1 , 2 , 3 は

```
n = int(input('n='))
m = int(input('m='))
d = 1
```

です。ステップ 4 は

```
while True:
    ステップ 4 の処理の中身
```

と無限ループを使います。while 文の条件が True と真ですから、break 文で抜け出す必要があります。ステップ 4 の (a) は n と m を共に割り切る g (1 以外のがあればその値を g とし、なければ g=1 とする) を返す 関数 `get_div(n, m)` を定義することで対処することにします。函数 `get_div(n, m)` の定義は

```
def get_div(n, m):
     for k in range(2, n+1):
         if n % k == 0 and m % k == 0:
            return k
     return 1
```

のようなもので良いです。これを使って、ステップ 4 の (a) は

```
    g = get_div(n, m)
    if g == 1: break
```

で実現できます。ステップ４の残りの部分は

```
    n = n // g
    m = m // g
    d = d * g
```

とします。// は整数を整数で割った時の商です。正確に言うと整数を整数で割った時の小数点以下を切り捨てたものです。ここで

```
    n = n / g
    m = m / g
```

と通常の割り算を使うと与えられた条件から割り切れるはずですが n と m が浮動小数点となってしまい、get_div(n, m) を呼び出した時、

```
     for k in range(2, n+1):
```

で、n は整数でないとエラー表示が出ます。従って、ステップ４は

```
while True:
    g = get_div(n, m)
    if g == 1: break
    n = n // g
    m = m // g
    d = d * g
```

となります。最後のステップ５は

```
print(' 最小公倍数 = ', d)
```

です。プログラム

```
def get_div(n, m):
     for k in range(2, n+1):
         if n % k == 0 and m % k == 0:
            return k
     return 1
n = int(input('n='))
m = int(input('m='))
d = 1
while True:
    g = get_div(n, m)
    if g == 1: break
    n = n // g
    m = m // g
    d = d * g
print(' 最小公倍数 = ', d)
```

が出来上がりました。実行すると

```
n=120
m=96
最小公倍数 =  24
```

です。

最大公約数の最後のアルゴリズムとして、素因数分解を使ったアルゴリズムをプログラミングしてみましょう。

このアルゴリズムのために必要な関数は l = decomp(n) です。正整数 n のすべての素因数からなるリスト l を返す関数を作ります。

```
def decomp(n):
    l = []
    k = 2
    while n > 1:
        while n % k == 0:
            l.append(k)
            n = n // k
        k = k + 1
    return l
```

でよいです。

```
    l = []
```

でまずリスト l を空リスト [] に初期化し、

```
    k = 2
    while n > 1:
        while n % k == 0:
            l.append(k)
            n = n // k
        k = k + 1
```

で、k=2 から始め、k で n が割り切れるうちは l に k を追加し、k で割り切れなくなれば、k を増やし、n=1 となるまで繰り返しています。最後に

```
    return l
```

で結果のリストを返しています。この関数をチェックするプログラムを作ります。

```
def decomp(n):
    l = []
    k = 2
    while n > 1:
        while n % k == 0:
            l.append(k)
            n = n // k
        k = k + 1
```

```
    return l
n = int(input('n= '))
m = int(input('m= '))
ln = decomp(n)
lm = decomp(m)
print(ln, lm)
```

実行すると

```
n= 120
m= 96
[2, 2, 2, 3, 5] [2, 2, 2, 2, 2, 3]
```

です。次に必要な関数は二つのリストを引数に取り、その共通部分を返す関数です。即ち、[2, 2, 2 , 3] = intersect([2, 2, 2, 3, 5], [2, 2, 2, 2, 2, 3]) となる関数 intersect(ln, lm) を作ってみましょう。色々なアルゴリズムが考えられますが、ここではリストのメソッド count(探索する要素) を使ってみます。

```
>>> l = [2,2,2,3,5]
>>> l.count(2)
3
>>> l.count(5)
1
>>> l.count(7)
0
>>>
```

のようにリストに引数で指定した探索する要素が何個あるか教えてくれるメソッドです。このメソッドを使えば、関数 intersect(ln, lm) は

```
def intersect(ln, lm):
    l = []
    k = ln[0]
    for i in range(k, ln[len(ln)-1]+1):
        j = min(ln.count(i), lm.count(i))
        for _ in range(j):
            l.append(i)
    return l
```

で、定義できます。凝ったことを考えずに、自然に思いついたアルゴリズムをプログラミングすればいいと思います。チェックするプログラムは

```
def decomp(n):
    l = []
    k = 2
    while n > 1:
        while n % k == 0:
```

```
            l.append(k)
            n = n // k
        k = k + 1
    return l
def intersect(ln, lm):
    l = []
    k = ln[0]
    for i in range(k, ln[len(ln)-1]+1):
        j = min(ln.count(i), lm.count(i))
        for _ in range(j):
            l.append(i)
    return l
n = int(input('n= '))
m = int(input('m= '))
ln = decomp(n)
lm = decomp(m)
print(ln, lm)
l = intersect(ln, lm)
print(l)
```

です。実行すると

```
n= 120
m= 96
[2, 2, 2, 3, 5] [2, 2, 2, 2, 2, 3]
[2, 2, 2, 3]
```

です。最後に、リストに含まれる素数を全てかけ合わせれば最大公約数になります。

```
def decomp(n):
    l = []
    k = 2
    while n > 1:
        while n % k == 0:
            l.append(k)
            n = n // k
        k = k + 1
    return l
def intersect(ln, lm):
    l = []
    k = ln[0]
    for i in range(k, ln[len(ln)-1]):
        j = min(ln.count(i), lm.count(i))
        for _ in range(j):
            l.append(i)
```

```
    return l
n = int(input('n= '))
m = int(input('m= '))
ln = decomp(n)
lm = decomp(m)
l = intersect(ln, lm)
gcd = 1
for i in l:
    gcd *= i
print(gcd)
```

です。実行すると

```
n= 120
m= 96
24
```

です。

問題：二つの自然数 N と M を入力し、その最小公倍数を出力するプログラムを作れ。

解答例： ここでは最後のプログラムの [2, 2, 2 , 3] = intersect([2, 2, 2, 3, 5], [2, 2, 2, 2, 2, 3]) となる関数 intersect(ln, lm) を作り替えて、[2, 2, 2 , 2, 2, 3, 5] = union([2, 2, 2, 3, 5], [2, 2, 2, 2, 2, 3]) となる関数 union(ln, lm) を定義することにより、プログラミングしてみます。勿論、これは模範解答ではないです。

関数 union(ln, lm) は

```
def union(ln, lm):
    l = []
    k = min(ln[0], lm[0])
    for i in range(k, max(ln[len(ln)-1], lm[len(lm)-1])+1):
        j = max(ln.count(i), lm.count(i))
        for _ in range(j):
            l.append(i)
    return l
```

で、定義できます。従って、プログラムの全体は

```
def decomp(n):
    l = []
    k = 2
    while n > 1:
        while n % k == 0:
            l.append(k)
            n = n // k
        k = k + 1
    return l
def union(ln, lm):
```

```
    l = []
    k = min(ln[0], lm[0])
    for i in range(k, max(ln[len(ln)-1], lm[len(lm)-1])+1):
        j = max(ln.count(i), lm.count(i))
        for _ in range(j):
            l.append(i)
    return l

n = int(input('n= '))
m = int(input('m= '))
ln = decomp(n)
lm = decomp(m)
l = union(ln, lm)
lcm = 1
for i in l:
    lcm *= i
print(ln, lm, l)
print(lcm)
```

です。実行すると

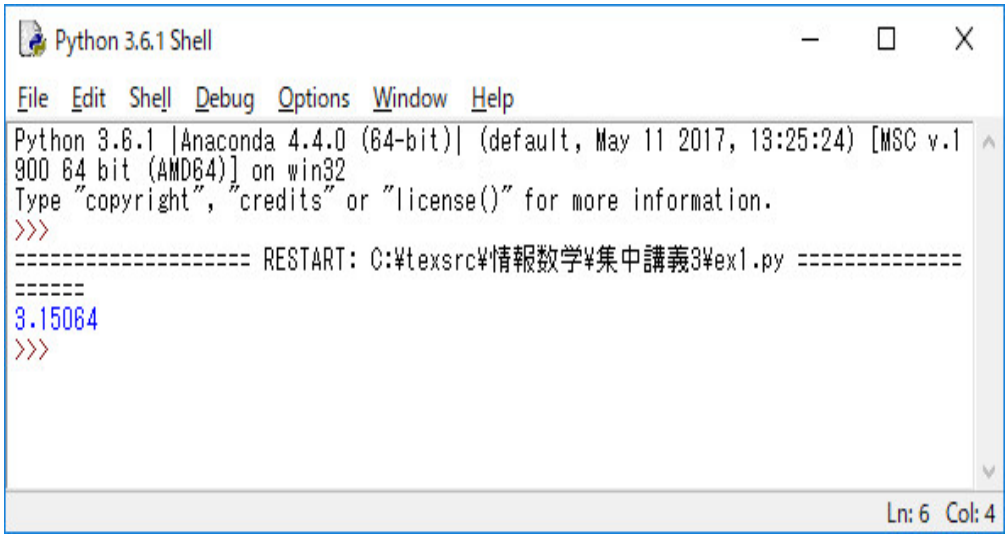
```
n= 120
m= 96
[2, 2, 2, 3, 5] [2, 2, 2, 2, 2, 3] [2, 2, 2, 2, 2, 3, 5]
480
```

です。

円周率はモンテカルロ法でも計算できます。正方形とそれに内接する円を描き、多数の点をランダムに置きます。円の中にある点の割合を調べれば円周率の近似値を求めることができます。素朴なプログラムは

```
import random
def getRatio(numThrows):
    count = 0;
    for t in range(numThrows):
        x = random.random()
        y = random.random()
        if x*x+y*y <= 1.0:
            count += 1;
    return 4*(count/float(numThrows))
print( getRatio(100000))
```

となります。実行すると



となります。。

```
import random
```

は乱数を使うために必要です。

```
    for t in range(numThrows):
```

は　t を 0 から numThrows - 1 まで変化させながら、以下のブロックを繰り返します。

```
        x = random.random()
        y = random.random()
```

で、 x と y に 0 から 1 の乱数をセットしています。したがって

```
    count = 0;
    for t in range(numThrows):
        x = random.random()
        y = random.random()
        if x*x+y*y <= 1.0:
            count += 1;
```

で、半径 1 の円の第一象限に (x,y) がある回数を count にカウントしています。最後に、

```
    return 4*(count/float(numThrows))
```

で、円周率の近似値を計算して、返します。
　実行するたびに答えが違います。このプログラムを多数回繰り返して、平均値を求めるともっと良い近似値が得られそうです。

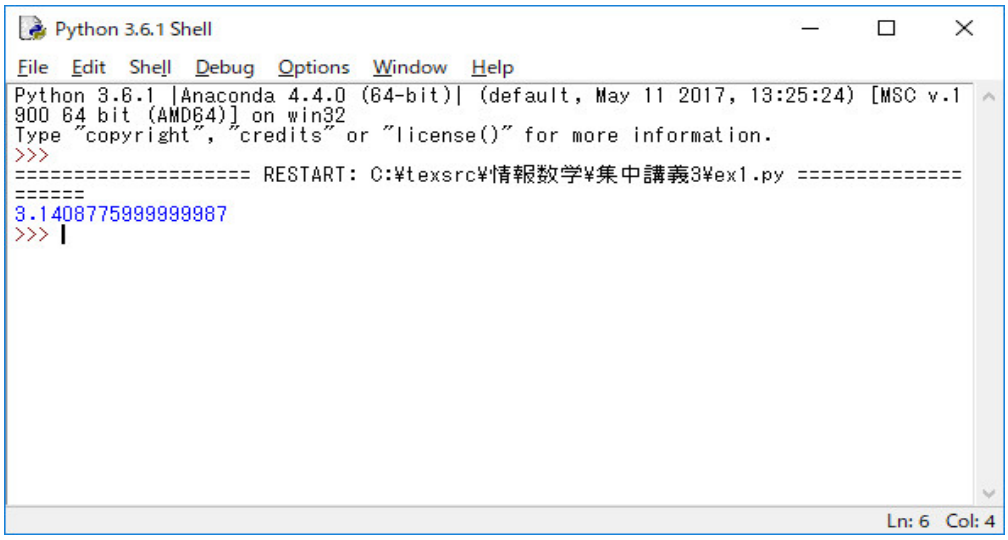
```
import random
def getRatio(numThrows):
    count = 0;
```

```
    for t in range(numThrows):
        x = random.random()
        y = random.random()
        if x*x+y*y <= 1.0:
            count += 1;
    return 4*(count/float(numThrows))
def getMean(numThrows, numTrials):
    estimates = []
    for t in range(numTrials):
        estimates.append(getRatio(numThrows))
    return sum(estimates)/len(estimates)
print( getMean(100000, 100) )
```

となります。新たに作った関数 getMean(numThrows, numTrials)

```
def getMean(numThrows, numTrials):
    estimates = []
    for t in range(numTrials):
        estimates.append(getRatio(numThrows))
    return sum(estimates)/len(estimates)
print( getMean(100000, 100) )
```

で、numTrials 回 getRatio(numThrows) を実行し、結果を estimates というリストに保存し、平均値を計算しています。実行すると

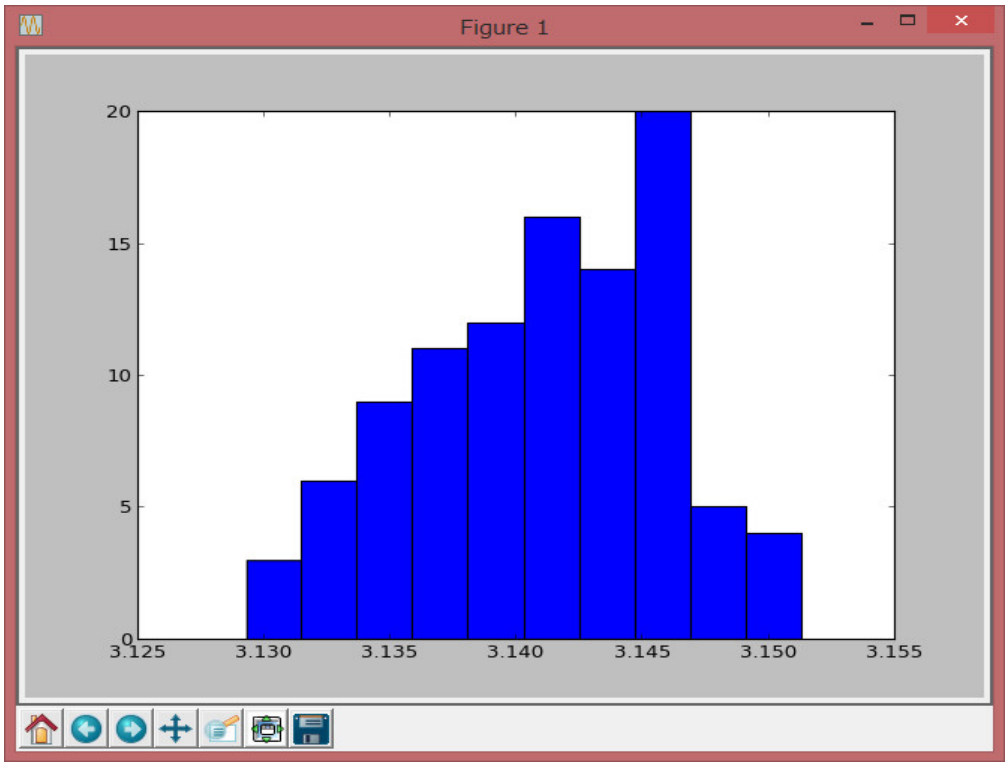


となります。我々は円周率のかなり良い近似値を知っていますから、表示された値がどれだけ良いか知っていますが、正確な答えを知らない一般の場合は、統計学を学んだ人は、円周率の近似値がどのように分布しているか、ヒストグラムを表示してみます。次のプログラムを実行するには pylab というライブラリーをインストールしなければなりません。

```
import random
```

```
import pylab

def getRatio(numThrows):
    count = 0;
    for t in xrange(numThrows):
        x = random.random()
        y = random.random()
        if x*x+y*y <= 1.0:
            count += 1;
    return 4*(count/float(numThrows))
def getMean(numThrows, numTrials):
    estimates = []
    for t in range(numTrials):
        estimates.append(getRatio(numThrows))
    pylab.hist(estimates, bins = 10)
    pylab.show()
    return sum(estimates)/len(estimates)
print getMean(100000, 100)
```

とプログラムを修正します。実行すると

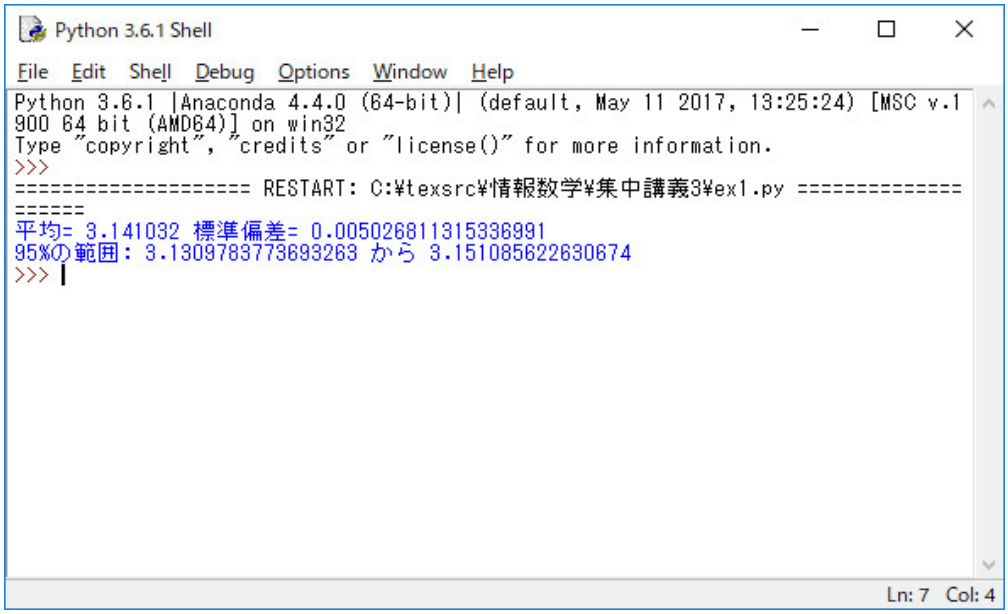


となります。

　このようにばらつきがあるので、正確な答えを知らない一般の場合は、統計学を学んだ人は、標準偏差も求め、円周率の存在範囲を推定します。

```
import random
def getRatio(numThrows):
    count = 0;
    for t in range(numThrows):
        x = random.random()
        y = random.random()
        if x*x+y*y <= 1.0:
            count += 1;
    return 4*(count/float(numThrows))
def getMean(numThrows, numTrials):
    estimates = []
    for t in range(numTrials):
        estimates.append(getRatio(numThrows))
    mean = sum(estimates)/len(estimates)
    var = 0.0
    for x in estimates:
        var += (x - mean)**2
    sdev = (var/(len(estimates)))**0.5
    return (mean, sdev)
mean , sdev = getMean(100000, 100)
print( u'平均=', mean, u'標準偏差=', sdev )
print( u'95%の範囲:', mean-2*sdev, u'から', mean+2*sdev  )
```

実行すると



となります。

単に統計学を表面的に学ぶのではなく、学んだことをこのように実際の問題に使ってみるとこにより、より良く理解できます。「知っていること」と「出来ること」との間には大きなギャップがあ

ります。単に「知っている」だけでは役に立ちません。

高等学校では相関係数も学びます。相関係数を計算してみましょう。そのためにはデータが必要です。世界の原子力発電所の数はたとえば、一般社団法人　日本原子力産業協会の資料があります。

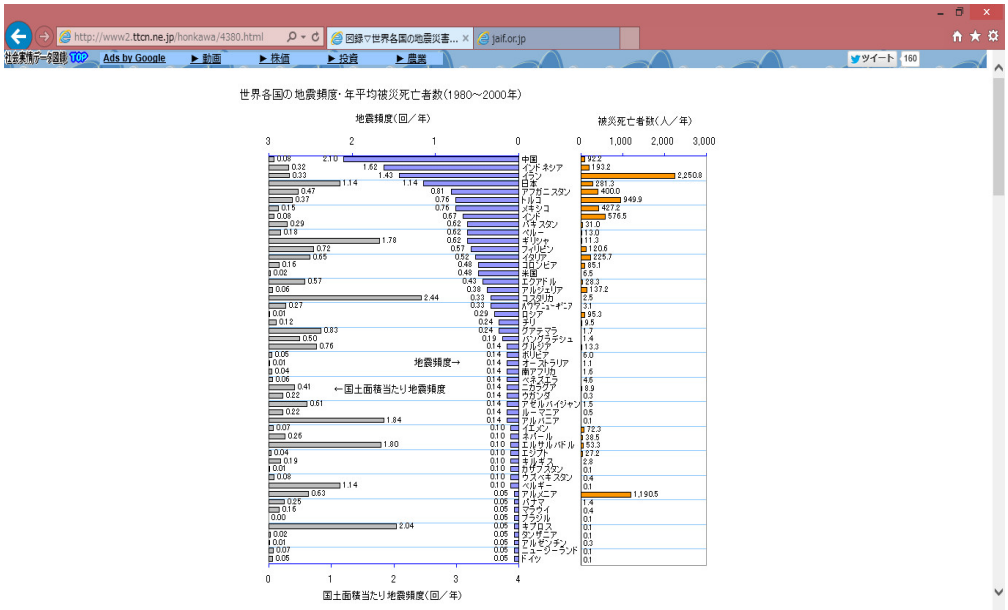
＜ 参 考 ＞

世界の原子力発電開発の現状

2014年1月1日現在、(万kW,グロス電気出力)
As of January 1, 2014 (10MWe, Gross Output)

国・地域		運転中 In Operation 出力 Output	運転中 In Operation 基数 Units	建設中 Under Construction 出力 Output	建設中 Under Construction 基数 Units	計画中 Planned 出力 Output	計画中 Planned 基数 Units	合計 Total 出力 Output	合計 Total 基数 Units	Country Region
1	米国	10,328.4	100	560.0	5	626.0	5	11,514.4	110	U.S.A.
2	フランス	6,588.0	58	163.0	1			6,751.0	59	France
3	日本*	4,426.4	48	442.1	4	1,158.2	8	6,026.7	60	Japan*
4	ロシア	2,519.4	29	1,026.0	11	1,744.5	17	5,289.9	57	Russia
5	韓国	2,071.6	23	660.0	5	560.0	4	3,291.6	32	Korea
6	中国	1,478.8	17	3,386.6	31	2,616.8	23	7,482.2	71	China
7	カナダ	1,424.0	19					1,424.0	19	Canada
8	ウクライナ	1,381.8	15	200.0	2			1,581.8	17	Ukraine
9	ドイツ	1,269.6	9					1,269.6	9	Germany
10	英国	1,086.2	16			326.0	2	1,412.2	18	United Kingdom
11	スウェーデン	942.8	10					942.8	10	Sweden
12	スペイン	739.7	7					739.7	7	Spain

国別の世界の地震の頻度のデータも探してみます。



この二つのデータからデータを作ります。

米国	110	0.16
フランス	59	
日本	60	1.14
ロシア	57	0.01
韓国	32	

中国	71	0.08
カナダ	19	
ウクライナ	17	
ドイツ	9	0.06
英国	18	
スウェーデン	10	
スペイン	7	
ベルギー	7	1.14
台湾	8	
インド	33	0.08
チェコ	8	
スイス	5	
フィンランド	7	
ブルガリア	3	
ハンガリー	4	
ブラジル	3	0.00
スロバキア	6	
南アフリカ	2	0.04
ルーマニア	5	0.22
メキシコ	2	0.15
アルゼンチン	3	0.01
イラン	2	0.33
パキスタン	7	0.29
スロベニア	1	
オランダ	1	
アルメニア	1	0.63
アラブ首長国連邦	4	
ベラルーシ	2	
トルコ	8	0.37
インドネシア	4	0.32
ベトナム	4	
バングラデシュ	2	0.50
エジプト	2	0.04
リトアニア	1	
ヨルダン	1	
イスラエル	1	
カザフスタン	1	0.01

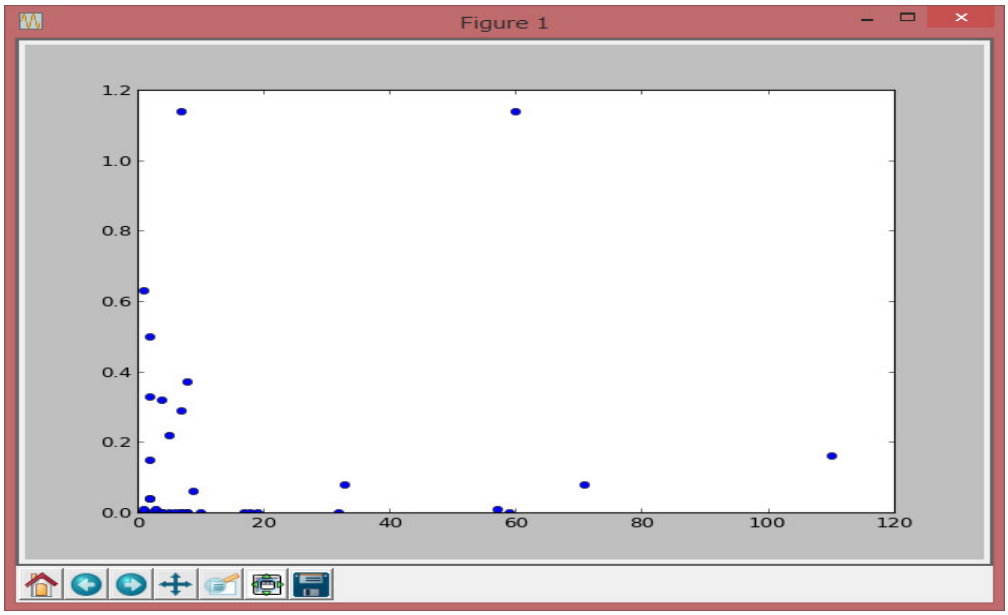
国名、運転中＋建設中＋計画中の原発の基数、日本の面積当たりマグニチュード 5.5 以上の地震頻度（回/年）です。単なる地震頻度（回/年）でなく、日本の面積当たりマグニチュード 5.5 以上の地震頻度（回/年）を使ったのは、国によって大きさが違い、大きな国は地震の発生場所が偏っているからです。地震頻度がないのはほとんど地震がなく上の表にデータがない国です。日本より地震頻度（回/年）が高い国が 5 か国ありますが原発はありません。これからリストを作ると

[[110, 0.16],[59, 0.0],[60, 1.14],[57, 0.01],[32, 0.0],[71, 0.08],[19, 0.0],[17, 0.0],[9, 0.06],[18, 0.0],[10, 0.0],[7, 0.0],[7, 1.14],[8, 0.0],[33, 0.08],[8, 0.0],[5, 0.0],[7, 0.0],[3, 0.0],[4, 0.0],[3, 0.0],[6, 0.0],[2, 0.04],[5, 0.22],[2, 0.15],[3, 0.01],[2, 0.33],[7, 0.29],[1, 0.0],[1, 0.0],[1, 0.63],[4, 0.0],[2, 0.0],[8, 0.37],[4, 0.32],[4, 0.0],[2, 0.50],[2, 0.04],[1, 0.0],[1, 0.0],[1,0.0],[1, 0.01]]

です。散布図を表示するには Pylab を使います。Pylab はデフォルトではインストールされていませんから、自分でインストールしてください。プログラムは

```
import pylab

l = [ [110, 0.16],[ 59, 0.0],[60, 1.14],[ 57, 0.01],[32, 0.0],[71, 0.08],
      [19, 0.0],[ 17, 0.0],[9, 0.06],[ 18, 0.0],[ 10, 0.0],[ 7, 0.0],
      [7, 1.14],[ 8, 0.0],[33, 0.08],[ 8, 0.0],[ 5, 0.0],[7, 0.0],
      [3, 0.0],[4, 0.0],[3, 0.0],[6, 0.0],[2, 0.04],[5, 0.22],[2, 0.15],
      [3, 0.01],[2, 0.33],[7, 0.29],[1, 0.0],[1, 0.0],[1, 0.63],[4, 0.0],
      [2, 0.0],[ 8, 0.37],[4, 0.32],[4, 0.0],[2, 0.50],[2, 0.04],
      [1, 0.0],[1, 0.0],[1,0.0],[1, 0.01]]
a = []
e = []
for x in l:
    a.append(x[0])
    e.append(x[1])

pylab.plot(a, e, 'bo')
pylab.show()
```

です。実行すると



です。横軸は原発の基数、縦軸は地震頻度（回/年）です。日本は上端中央です。

相関係数は

$$\text{相関係数} \quad = \frac{\text{共分散}}{x \text{ の標準偏差} \cdot y \text{ の標準偏差}}$$

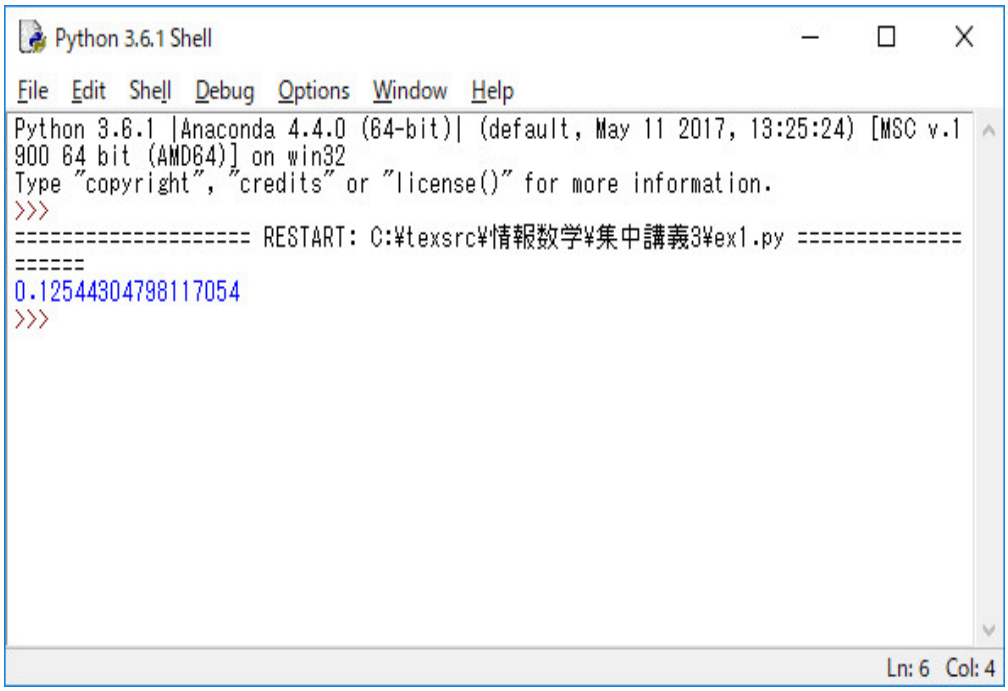
です。共分散は

$$\text{共分散} = \frac{1}{n}\{(x_1 - \overline{x})(y_1 - \overline{y}) + \cdots + (x_n - \overline{x})(y_n - \overline{y})\}$$

です。したがって、プログラムは

```
def mean(xs):
    m = 0.0
    for x in xs:
        m += x
    return m / len(xs)
def std(xs):
    m = mean(xs)
    s = 0.0
    for x in xs:
        s += (x-m)**2
    s /= len(xs)
    return s ** 0.5
def covariance(xs, ys):
    mx = mean(xs)
    my = mean(ys)
    s = 0.0
    for i in range(len(xs)):
        s += (xs[i]-mx)*(ys[i]-my)
    return s / len(xs)
def correlation(xs, ys):
    return covariance(xs, ys)/(std(xs) * std(ys))

l = [ [110, 0.16],[ 59, 0.0],[60, 1.14],[ 57, 0.01],[32, 0.0],[71, 0.08],
      [19, 0.0],[ 17, 0.0],[9, 0.06],[ 18, 0.0],[ 10, 0.0],[ 7, 0.0],
      [7, 1.14],[ 8, 0.0],[33, 0.08],[ 8, 0.0],[ 5, 0.0],[7, 0.0],
      [3, 0.0],[4, 0.0],[3, 0.0],[6, 0.0],[2, 0.04],[5, 0.22],[2, 0.15],
      [3, 0.01],[2, 0.33],[7, 0.29],[1, 0.0],[1, 0.0],[1, 0.63],[4, 0.0],
      [2, 0.0],[ 8, 0.37],[4, 0.32],[4, 0.0],[2, 0.50],[2, 0.04],
      [1, 0.0],[1, 0.0],[1,0.0],[1, 0.01]]
a = []
e = []
for x in l:
    a.append(x[0])
    e.append(x[1])

print( correlation(a, e) )
```

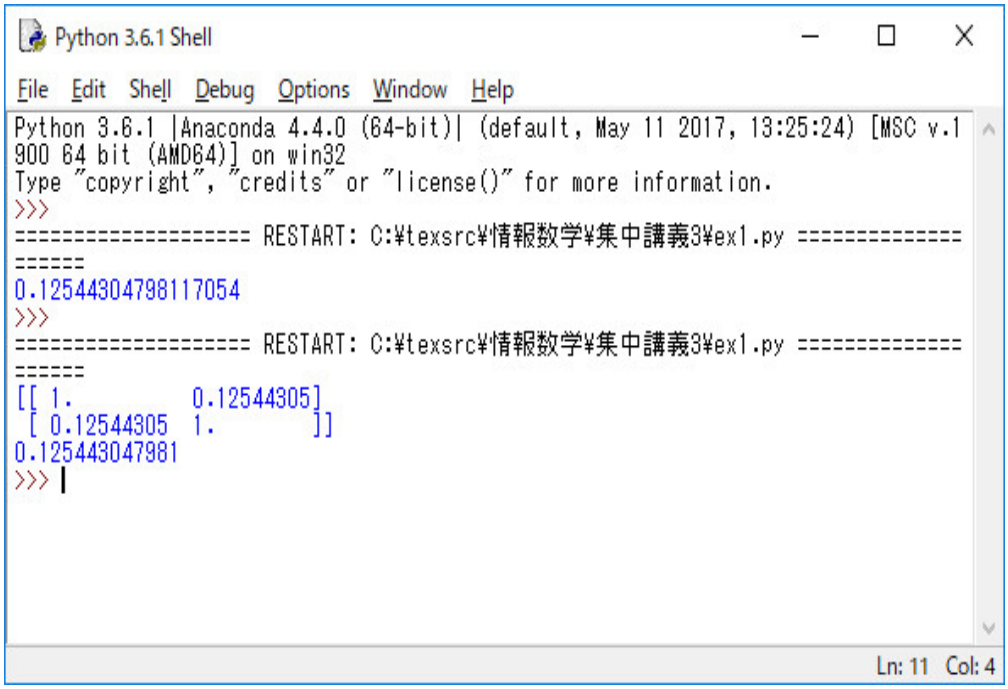
で、実行すると



です。numpy を使うと

```
import numpy

l = [ [110, 0.16],[ 59, 0.0],[60, 1.14],[ 57, 0.01],[32, 0.0],[71, 0.08],
      [19, 0.0],[ 17, 0.0],[9, 0.06],[ 18, 0.0],[ 10, 0.0],[ 7, 0.0],
      [7, 1.14],[ 8, 0.0],[33, 0.08],[ 8, 0.0],[ 5, 0.0],[7, 0.0],
      [3, 0.0],[4, 0.0],[3, 0.0],[6, 0.0],[2, 0.04],[5, 0.22],[2, 0.15],
      [3, 0.01],[2, 0.33],[7, 0.29],[1, 0.0],[1, 0.0],[1, 0.63],[4, 0.0],
      [2, 0.0],[ 8, 0.37],[4, 0.32],[4, 0.0],[2, 0.50],[2, 0.04],
      [1, 0.0],[1, 0.0],[1,0.0],[1, 0.01]]
a = []
e = []
for x in l:
    a.append(x[0])
    e.append(x[1])
corr = numpy.corrcoef(a, e)
print( corr )
print( corr[0, 1] )
```

で、実行すると



です。相関はありません。numpy を使うと簡単ですが、Python で上のようなプログラムが自由に組めるようになってから、numpy を使うのが良いです。

もう一つ相関係数の例題を考えてみます。

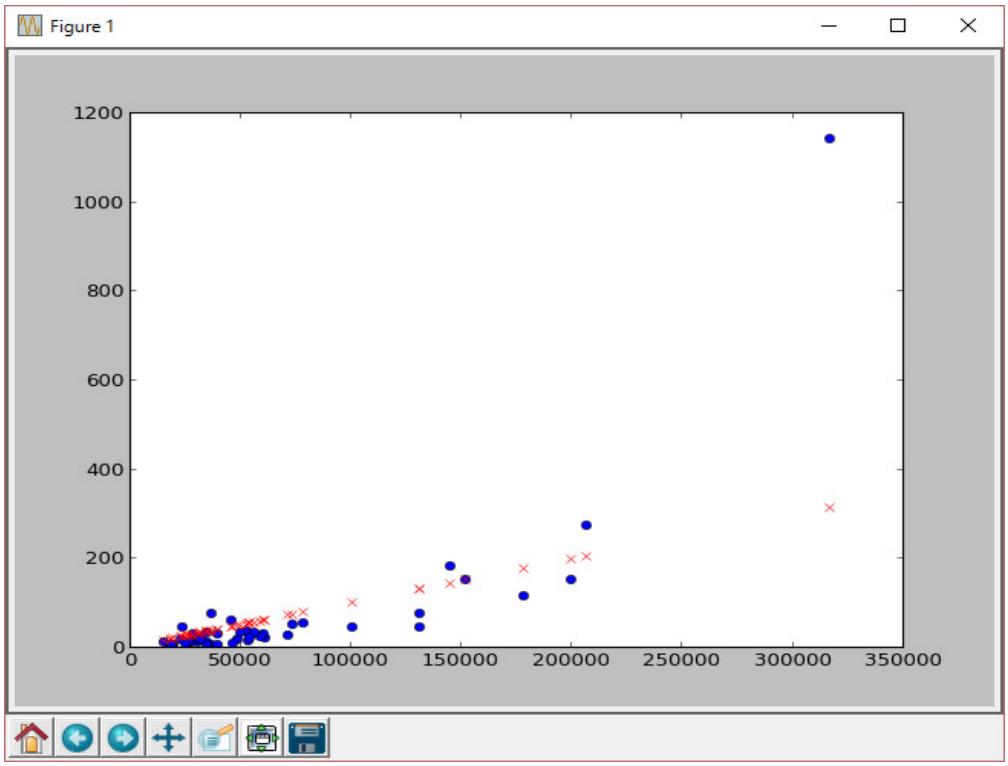
```
import pylab
import numpy

l = [[46, 131375],[5, 37127],[13, 35134],[20, 61264],[10, 26260],
     [13, 31103],[15, 53624],[55, 78920],[28, 54035],[35, 53017],
     [114, 178021],[152, 152338],[1143, 316830],[273, 206428],
     [30, 60385],[29, 28416],[21, 32325],[17, 22949],[8, 26059],
     [24, 58983],[32, 56432],[44, 100510],[152, 200067],[32, 50044],
     [6, 39755],[27, 71985],[46, 23657],[183, 145174],[77, 37311],
     [13, 27974],[12, 15300],[7, 18675],[22, 54527],[52, 73550],
     [7, 34840],[6, 19664],[13, 26488],[32, 35304],[5, 19229],
     [75, 131285],[9, 25317],[31, 40008],[19, 48551],[17, 31760],
     [19, 32119],[60, 45855],[8, 46746]]
a = []
e = []
for x in l:
    a.append(x[0])
    e.append(x[1])
corr = numpy.corrcoef(e, a)
print corr[0, 1]
N = sum(a)
M = sum(e)
```

```
b = []
for x in e:
   b.append(float(x)*N/M)
pylab.plot(e, a, 'bo')
pylab.plot(e, b, 'rx')
pylab.show()
```

実行すると

```
Python 2.7.9 (default, Dec 10 2014, 12:24:55) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
0.795783420928
>>>
```

と



となります。横軸は平成２７年８月１２日現在の都道府県別高校生数、青丸に対する縦軸は平成２８年度の都道府県別東大合格者数で、赤×に対する縦軸は都道府県別高校生数に対して東大合格者数を比例配分したときの都道府県別東大合格者数です。計算上は、相関係数は　0.795783420928　で高い相関があると言えますが、右上隅が東京都で、合格者の大部分を占めていて、直線的な分布にはなってなく、極端な外れ値になっています。**単純に相関係数を計算するだけでなく、散布図を描いて可視化し、解析することが必要です。**

演習問題

1. 自然数 n を入力し n を 3 で割ったときの商と余りを表示するプログラムを作れ。
2. ４桁の数を入力して千、百、十、一の位の数を出力するプログラムを書け。
3. 対数関数 LOG() または log10(x) を利用して入力した数の桁数を出力するプログラムを書け。
4. 余弦定理 $c^2 = a^2 + b^2 - 2ab\cos C$ を利用して、a,b,C の値から三角形の辺 AB の長さ c を求めるプログラムを作れ。
5. 三角形 ABC の辺 BC の長さ a と角 A から、外接円の半径 R を求めるプログラムを作れ
6. ボールの初速 (m/s) と投げる角度を入力し、ボールの到達距離を計算するプログラムをつくれ。高等学校の物理の問題です。

旺文社の昔の数学 A の教科書には、次のような最大公約数を求めるプログラムが載っています。

```
10 REM 最大公約数
20 INPUT "N, M"; N, M
30 G=N
40 IF N>INT(N/G)*G THEN 80
50 IF M>INT(M/G)*G THEN 80
60 PRINT "最大公約数＝";G
70 GOTO 100
80 G=G-1
90 GOTO 40
100 END
```

これは昔のプログラミングスタイルで、読みにくいプログラムになります。BASIC は各行に順番に行番号を付けます。

```
10 REM 最大公約数
```

はコメントです。BASIC はこの行を無視します。

```
20 INPUT "N, M"; N, M
```

はキーボードから数値 N, M を入力します。Python では

```
n = int(input('n='))
m = int(input('m='))
```

に当たります。

```
30 G=N
```

は変数 G に変数 N の値を代入します。

```
40 IF N>INT(N/G)*G THEN 80
```

は、N が G で割り切れないなら８０行にジャンプしなさいという IF 文です。$INT(N/G)$ で N を G で割って、小数点以下を切り捨てます。そして、$INT(N/G) * G$ で G 倍するわけですから、N が G で割り切れなければ $INT(N/G) * G$ は N より小さくなり、N が G で割り切れれば $INT(N/G) * G$ は N と等しくなります。

```
50 IF M>INT(M/G)*G THEN 80
```

は、M が G で割り切れないなら８０行にジャンプしなさいという IF 文です。

```
60 PRINT "最大公約数＝";G
```

で、文字列 ”最大公約数＝”と。変数 G の値を表示します

```
70 GOTO 100
```

は、行番号 100 の行に制御を移しなさい（次は 100 行を実行しなさい）という命令です。GOTO 文と言います。

```
80 G=G-1
```

は、G の値を一つ引きなさいと言う命令です。

```
90 GOTO 40
```

は、行番号 40 の行に制御を移しなさい（次は 40 行を実行しなさい）という命令です。

```
100 END
```

は、プログラムが終了したことを示す命令です。

このプログラムのように goto 文を多用すると「スパゲッチイプログラム」という読みにくいプログラムになるので、 goto 文有害論が出てきて、goto 文を使わなくてもプログラムが作れる PASCAL や C と言った構造的プログラミング言語が作られました。更に、Windows の普及とともに、「使って天国、作って地獄」という状況を緩和するためオブジェクト指向言語である C++ や Java や Python や Ruby といった言語が普及してきます。BASIC は膨大なプログラムの蓄積はありますが、二世代前のプログラミング言語です。

このプログラムが気になるなら私の Java による BASIC インタプリタのホームページをみて、その解説やサンプルプログラムを読んでください。センター試験の BASIC の問題はこの BASIC のプログラムの読みにくさが問題の出題意義になっています。BASIC は死んだ言語です。ただ、非常に特殊な問題に興味があり、その解法を解説している本が２０年前とかの古い本で BASIC で説明しているものしかなければ、BASIC を学んで勉強するしかないですが、普通の小中高の数学の先生になる人は、もはや BASIC のような過去の言語の勉強は害になるだけです。ちなみに、昔の BASIC の本には簡単なキャラクターベースのゲームのプログラミングを解説している本とか種々のパズルの解法を解説している本とか常微分方程式の解曲線を表示するプログラムを解説している本など興味深い本がありました。最近はそのような本は私が気づかないだけかもしれませんが見かけません。 すべて絶版になっています。BASIC の新しい本は需要が見込めないので多分今後出版されないと思います。今、多く出版されているのは C・C++ と Java の本です。それに Python や Ruby の本が続きます。また、関数型プログラミング関連の Haskell, Clojure, Scala, ErLang といった書籍もあります。本屋に行ってコンピュータ関連の棚を眺めてみると人々がいま何に関心があるかわかります。インターネットでは種々のパズルの解法を Prolog や Scheme や Python や Haskell でプログラミングしているホームページがあります。BASIC の英語の意味から BASIC はコンピュータ言語の「基礎」だと誤解している数学教育の先生も過去にはいましたが、BASIC は実際は Biginer’s all purpose symbolic instruction code の略で、Fortran が難しくて理解できない初心者のために開発されたコンピュータ言語で、１９８０年頃、BASIC を売り出されたばかりの

マイコン（昔のパソコン）の OS 兼言語としてビル・ゲイツとその友人が開発し売り出し、一時期、マイコンを使うみんなが使っていたものですが、Windows のプログラミングが当たり前になり、 C++ や Python などそれまでの言語のいいところを取り込んで改良された高機能で使いやすい言語が開発されている今の時代には BASIC は時代遅れの代物です。C++ や Java は会社に就職するには有利な言語ですがプログラミング初心者には習得が困難です。Python は BASIC なみに習得が容易で、基本を習得した後の利用価値が全然違います。Ruby も Python と同じようなオブジェクト指向言語ですが、同じプログラムを何通りにもプログラミングする方法があり、プログラミングが楽しくなる言語ですが、Python より習得が難しいです。昔ながらの BASIC と違って、Fortran は数値計算の膨大なプログラムの蓄積があるので、新しい機能を取り込みながら Fortran の改良が続けられ、使われ続けています。数値計算に興味があるなら、gfortran というフリーな Fortran のコンパイラが Windows10　でも使えます。32 ビットの OS なら MinGW、６４ビットの OS なら MinGW-w64 をインストールすれば使えます。Cygwin のようにインストールや使い方が難しいことはなく、文字コードは Shift-JIS です。 MinGW-w64 をインストールすれば、gcc や g++ もコマンドプロンプトでの使用ですが使えます。

歴史的に多分一番古いアルゴリズムはエジプト乗法アルゴリズムです。これはたとえば 41×59 を計算するのに、４１を２で順に割って行って余りが１になるところを見つけ（現代の言葉では２進数に変換し）、

$$41 \times 59 = (1 \times 59) + (8 \times 59) + (32 \times 59)$$

と計算します。さらに

$$8 \times 59 = 4 \times (59+59) = 2 \times ((59+59)+(59+59)) = ((59+59)+(59+59))+((59+59)+(59+59))$$

のように計算します。

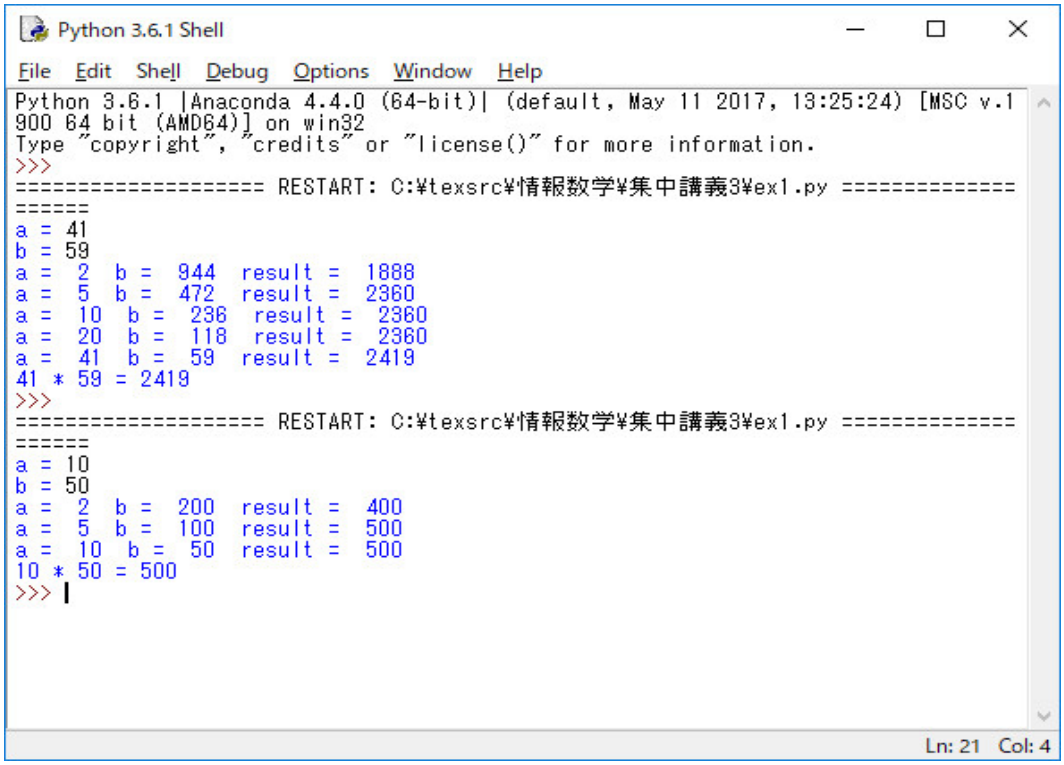
これを Python でプログラミングすると

```
def multiply(a, b):
    if a == 1:
        return b
    result = multiply(a//2, b+b)
    if a % 2 == 1:
        result = result + b
    print( "a = ", a, " b = ", b, " result = ", result)
    return result
a = int(input("a = "))
b = int(input("b = "))
c = multiply(a, b)
print( a, "*", b, "=", c)
```

となります。途中結果がわかるように

```
    print( "a = ", a, " b = ", b, " result = ", result)
```

と print 文を入れています。実行すると



です。

次の例は素因数分解の BASIC のプログラムです。これも旺文社の昔の数学 A の教科書で見つけたプログラムです。

```
10 REM 素因数分解
20 INPUT N
30 M=1
40 M=M+1
50 IF INT(N/M)*M<N THEN 40
60 PRINT M;
70 N=N/M
80 IF N>1 THEN 50
90 END
```

これも次のようにプログラミングするのが良いです。

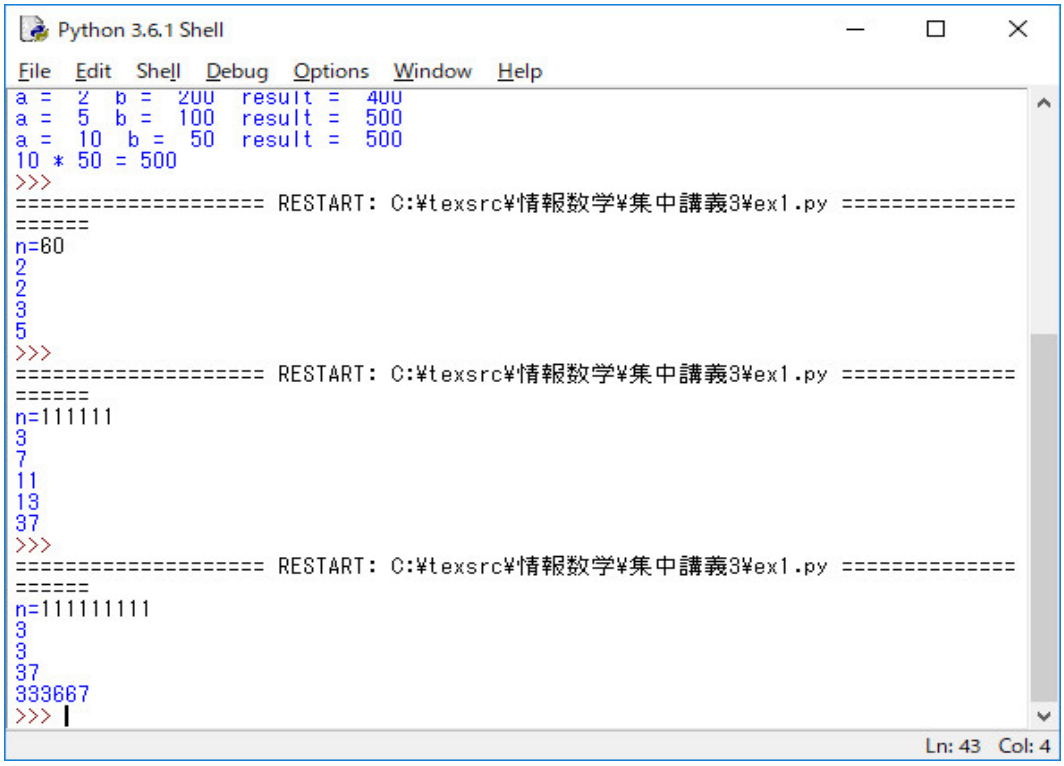
```
n = int(input('n='))
k = 2
while n > 1:
    while n % k == 0:
        print( k )
        n = n // k
    k = k + 1
```

while 文が入れ子になっています。

ここで使われているアルゴリズムは

1. N を入力する
2. K を 2 とする
3. N が 1 より大きい間は次を繰り返す
   (a) N が K で割り切れる間は K を表示し、N を K で割る
   (b) K をひとつ増やす

です。実行してみます。




```
n = int(input('n='))
k = 2
while n > 1:
    while n % k == 0:
        print( k , end='  ')
        n = n // k
    k = k + 1
```

とすると

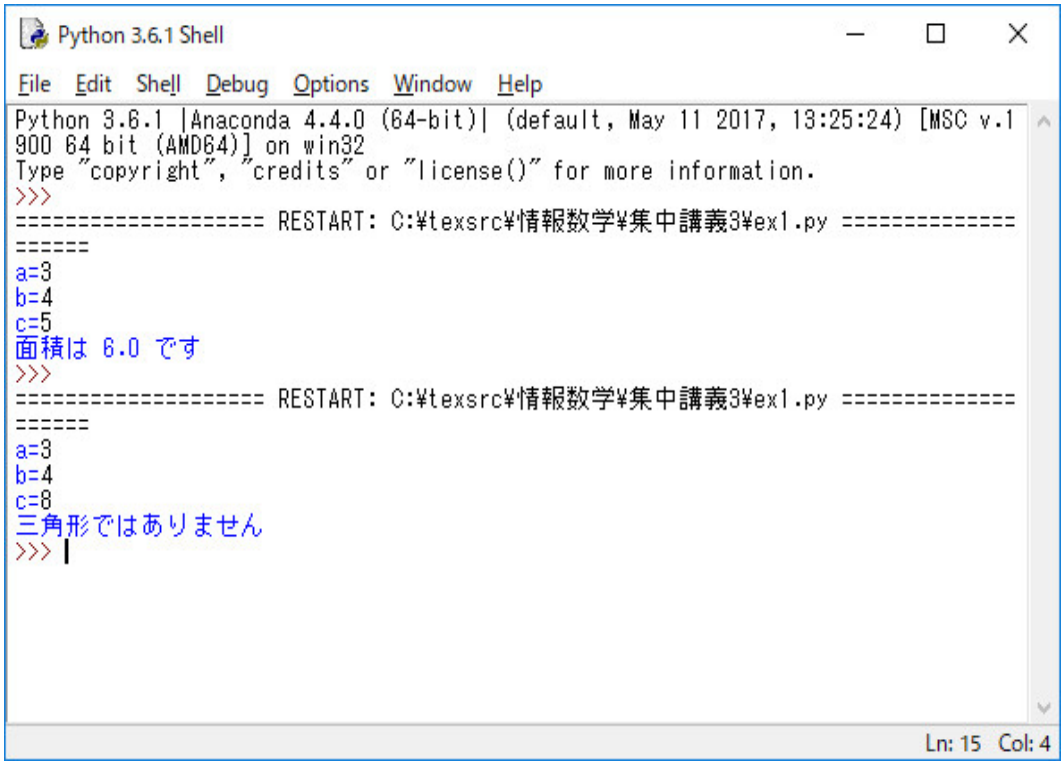
となります。

プログラムを作るときは、何をすべきか箇条書きにし、それをだんだんプログラムコードに近づけていきます。まず何をすべきか言葉で表現し、それを実現するには何をすればいいか箇条書きにし、そのそれぞれのステップを実現するには何をすればいいかそれぞれ箇条書きにし、を繰り返し、プログラムに近づけます。これをトップダウンの方法と言います。他人の書いたプログラムを読むときは各ステップが何をやっているか解析し、まとまりのあるいくつかのステップが何をやっているか解析し、段々大きく括って何をしているか調べ、最終的に何をしているか調べます。これをボトムアップの方法と言います。

例題：ヘロンの公式を使って、三角形の面積を計算するプログラムを作れ。

```
import math
import sys

a = float(input('a='))
b = float(input('b='))
c = float(input('c='))
if a+b <= c or b+c <= a or c+a <= b :
    print(  "三角形ではありません" )
    sys.exit()
S = (a+b+c)/2.0
S = math.sqrt(S*(S-a)*(S-b)*(S-c))
print( "面積は",  S, "です" )
```

実行してみます。




```
if a+b <= c or b+c <= a or c+a <= b :
    print(  "三角形ではありません" )
    sys.exit()
```

の if 文は、$a+b$ が　c　以下（a+b<=c) か $b+c$ が　a　以下（b+c<=a) か $c+a$ が　b　以下（c+a<=b) の時という意味です。「または」を表す論理演算子は or です。以下と以上を表す比較演算子は<= と >= です。否定を表す論理演算子は not です。

```
    if a+b<=c or b+c<=a or c+a<=b :
```

は

```
    if not (a+b > c and b+c > a and c+a> b) :
```

と表現することも出来ます。

```
    sys.exit()
```

は、プログラムを終了します。これを使うためには

```
import sys
```

が必要です。

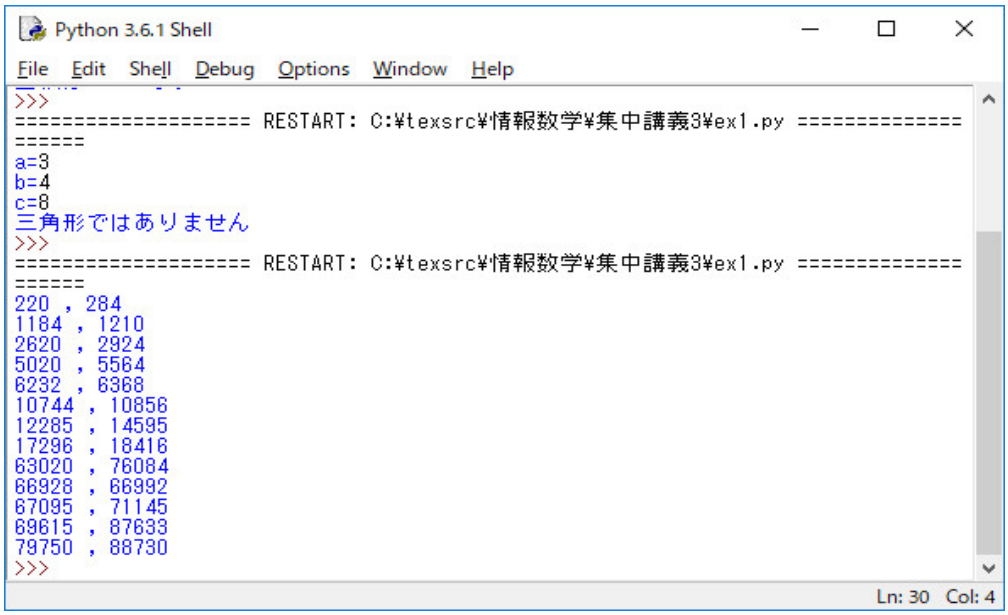
例題：100000 以下の友愛数をすべて求めてみよう。友愛数（ゆうあいすう）とは、異なる 2 つの自然数の組で、自分自身を除いた約数の和が、互いに他方と等しくなるような数をいう。親和数とも呼ばれる。

一番小さな友愛数の組は (220, 284) である。220 の自分自身を除いた約数は、1,2,4,5,10,11,20,22,44,55,110 で、和は 284 となる。一方、284 の自分自身を除いた約数は、1,2,4,71,142 で、和は 220 である。

プログラムの例は

```
def sum(n):
    s = 0
    for k in range(1, n//2+1):
        if n % k == 0: s = s + k
    return s
for i in range(1, 100001):
    s = sum(i)
    if s <= i: continue
    if i == sum(s) :
        print( i, ',', s )
```

です。実行する。



Python はインタプリッタなのでさすがに時間がかかります。一回限りでない、他人に使って貰うプログラムは C や C++ のようなコンパイラを使います。

```
def sum(n):
    s = 0
    for k in range(1, n//2+1):
        if n % k == 0: s = s + k
    return s
```

が、整数 n の自分自身以外の約数の和を計算する関数です。

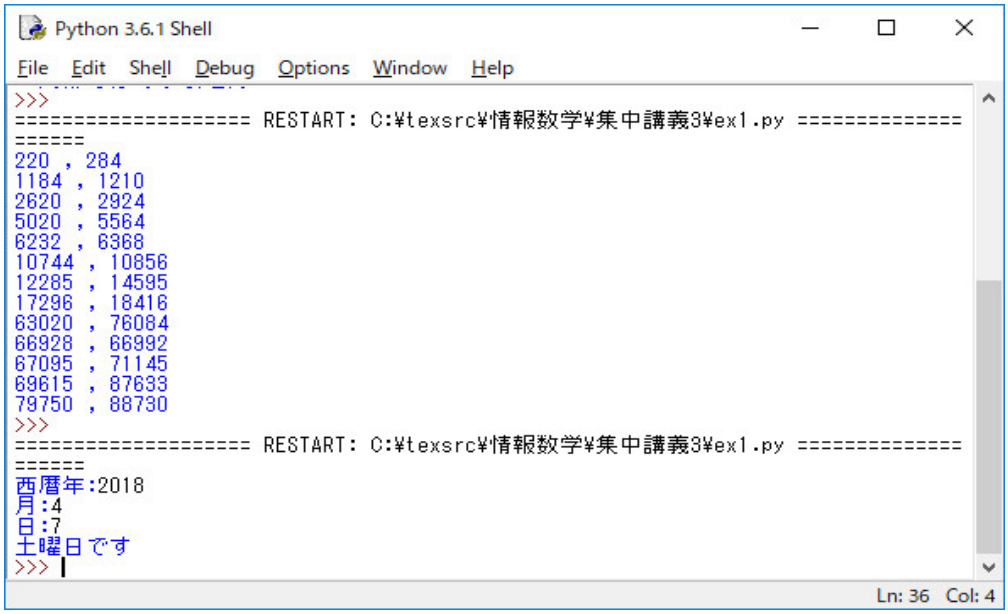
問題： **100000** 以下の完全数を計算するプログラムを作れ。

例題：**Zeller** の公式を使って、西暦の年月日を入力して曜日を計算するプログラムを作れ。

```
y = int(input("西暦年:"))
m = int(input("月:"))
d = int(input( "日:"))
```

```
m = m-2
if m < 1 :
    y = y-1
    m = m + 12
b = y % 100
c = y // 100
w = ((26*m-2)//10+d+b+b//4+5*c+c//4) % 7
if w == 0:
    print(  "日曜日です" )
elif w == 1:
    print(  "月曜日です" )
elif w == 2:
    print(  "火曜日です" )
elif w == 3:
    print(  "水曜日です" )
elif w == 4:
    print(  "木曜日です" )
elif w == 5:
    print(  "金曜日です" )
elif w == 6:
    print(  "土曜日です" )
```

実行する。



理由は良く分からないけど正しいそうです。このアルゴリズムは 1980 年頃のプログラミングの本で読んだ記憶があります。昔、数学者の一松信さんが解説を書いていましたが良く理解できませんでした。与えられた日の曜日はこのアルゴリズムを使わなくても曜日の分かっている日からの日数を計算することによって判定することが出来ます。

演習問題

1. 二つの整数 n, m を入力し、その最小公倍数を計算するプログラムを作れ。
2. ある都市の中型タクシーの料金は、2 km までは 500 円、あとは 300 m までごとに 80 円増すことになっている。乗車距離 x (km) を入力し、料金 y (円) を表示するプログラムを作れ。
3. 西暦年を入力して閏年かどうかを判定するプログラムを作れ。１５８３年以降グレゴリオ暦が使われていて、閏年は、４で割り切れるが１００の倍数で無いときである。ただし４００で割り切れれば閏年である。
4. a,b,c が定数、$a \neq 0$ のとき a,b,c を入力して、二次方程式 $ax^2 + bx + c = 0$ の解の個数を求めるプログラムを作れ。
5. 三角形 ABC の３辺の長さ a, b, c を与えこれから三角形 ABC が鋭角三角形、直角三角形、鈍角三角形のいずれであるかを判定するプログラムを作れ。

例題：整数を入力し、素数かどうかを判定するプログラムを作れ。
これは色々な作り方がありますが、単純なアルゴリズムは次のようなものです。

1. N を入力する
2. I を 2, 3, 4, 5, $\cdots$, と $\sqrt{N}$ まで変化させ、I が N を割り切るなら N は合成数
3. 上のチェックで合成数でなければ素数

これをもっと具体的にします。
N を入力するは

```
n = int(input('n='))
```

で表現できます。
I を 2, 3, 4, 5, $\cdots$, と $\sqrt{N}$ まで変化させ、
は for 文で表現できます。

```
for i in range(2, int(math.sqrt(n))+1):
```

とすれば良いです。sqrt() を使うために math を import します。range() に与える数値は整数でないと駄目なので、 int(math.sqrt(n)) と実数値 math.sqrt(n) を int() で整数値に変換します。
I が N を割り切るならは if 文を使います。

```
    if  n % i == 0 :
```

です。
従って N は合成数の部分は

```
    if  n % i == 0 :
        print( n, 'は', i, 'で割り切れます' )
        sys.exit()
```

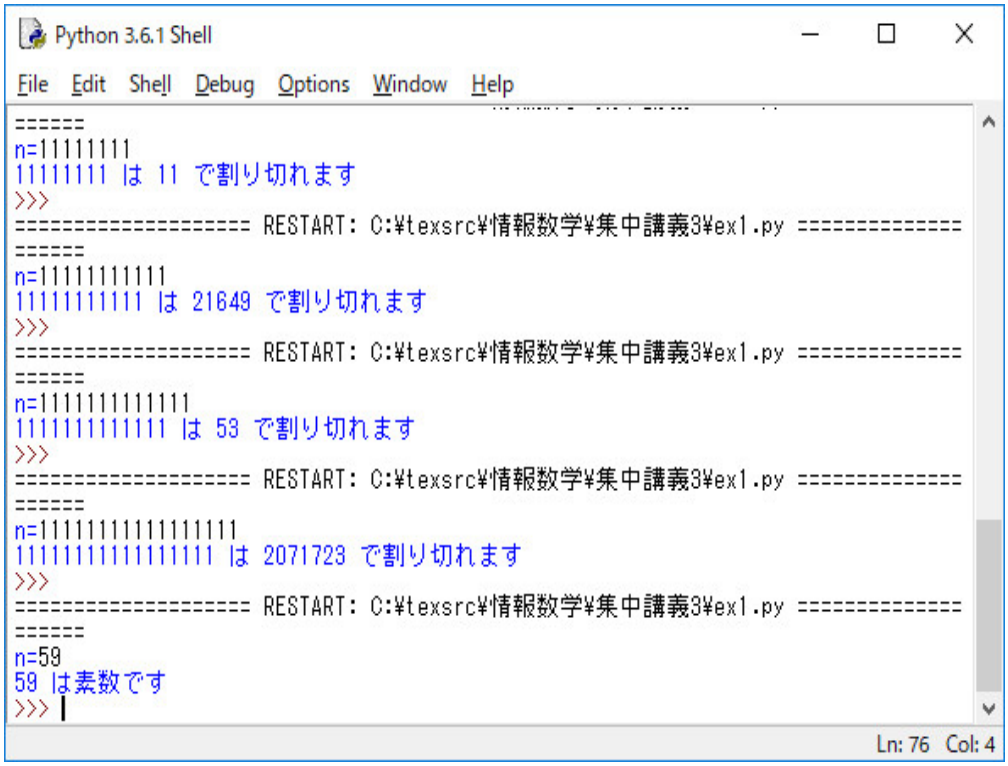
です。
上のチェックで合成数でなければ素数の部分は

```
print( n,  "は素数です" )
```

です。これらをまとめると

```
import math
import sys

n = int(input('n='))
for i in range(2, int(math.sqrt(n))+1):
    if  n % i == 0 :
        print( n, 'は', i, 'で割り切れます' )
        sys.exit()
print( n,  "は素数です" )
```

プログラムはこれで完成です。
実行する。



これは次のように関数を使ってもいいです。

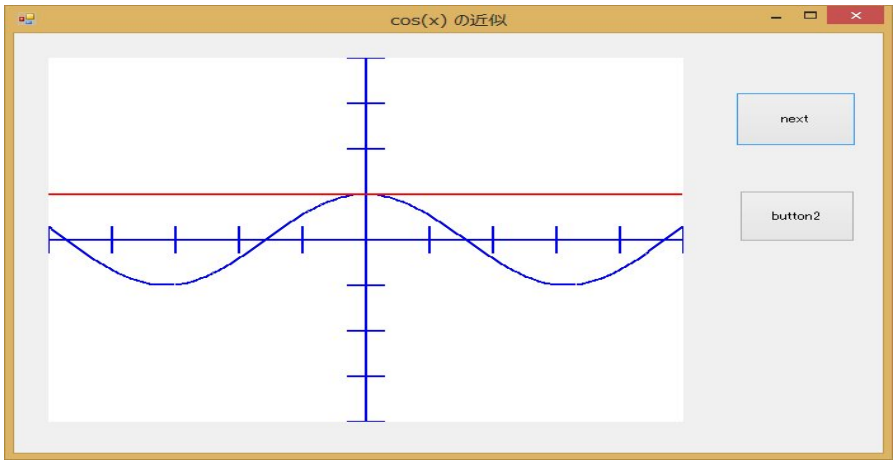
```
import math

def primeP(n) :
    for i in range(2, int(math.sqrt(n))+1):
```

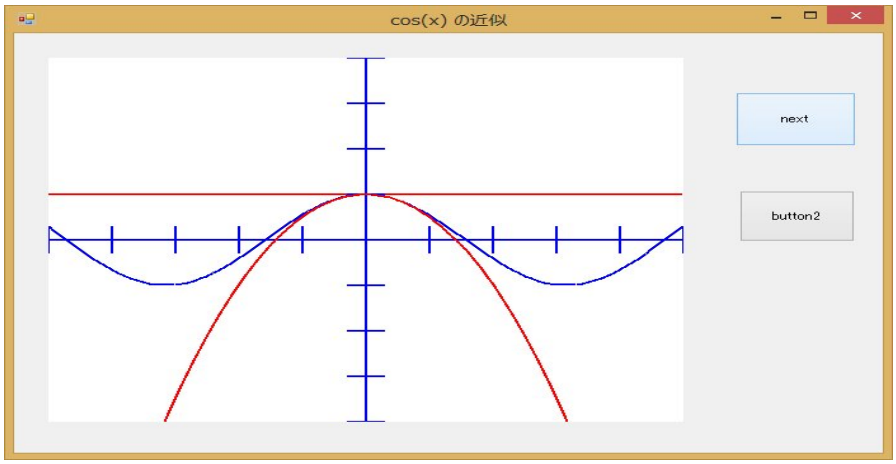
```
        if n % i == 0 :
            return i
    return 0
n = int(input('n='))
p = primeP(n)
if p == 0 :
    print( n,  "は素数です" )
else :
    print( n, "は合成数です", p, "で割れます" )
```

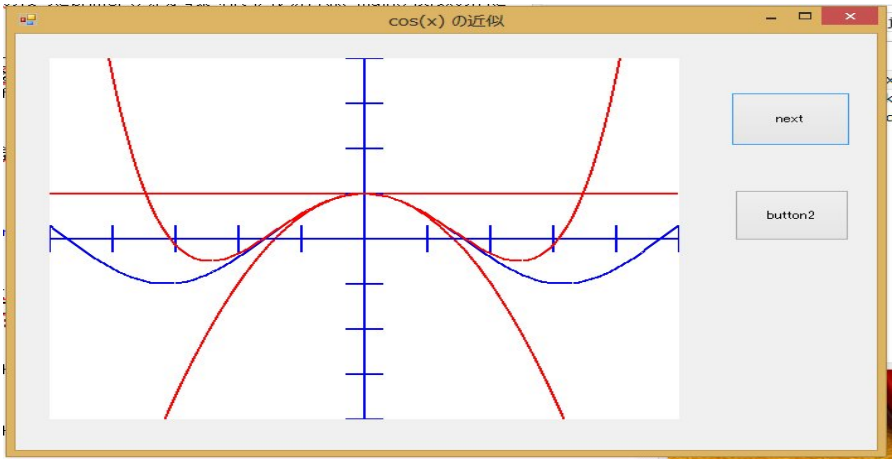
例題： 解析学で学ぶように、$\cos(x)$ や $\sin(x)$ は

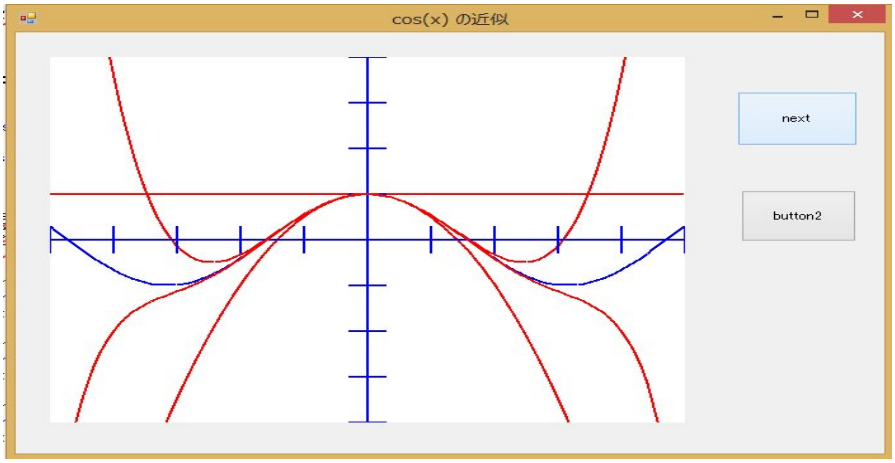
$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n}, \qquad \sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

と級数を使って表すことができます。x はラジアンです。この級数はすべての x に対して定義されていて非常に計算効率が良いです。これを使って、$\cos(x)$ を計算するプログラムを作れ。









のように

$$cosN(N,x) = \sum_{n=0}^{N} \frac{(-1)^n}{(2n)!} x^{2n}$$

は　$\cos(x)$ に近づいていきます。$\cos(x)$ を

$$cosN(N,x) = \sum_{n=0}^{N} \frac{(-1)^n}{(2n)!} x^{2n}$$

で近似するわけですが、x により N の値をなんにすればよいかが決まります。

それで、 N の値を決めるには、例えば、

$$|cosN(N,x) - cosN(N-1,x)| < 0.0000001$$

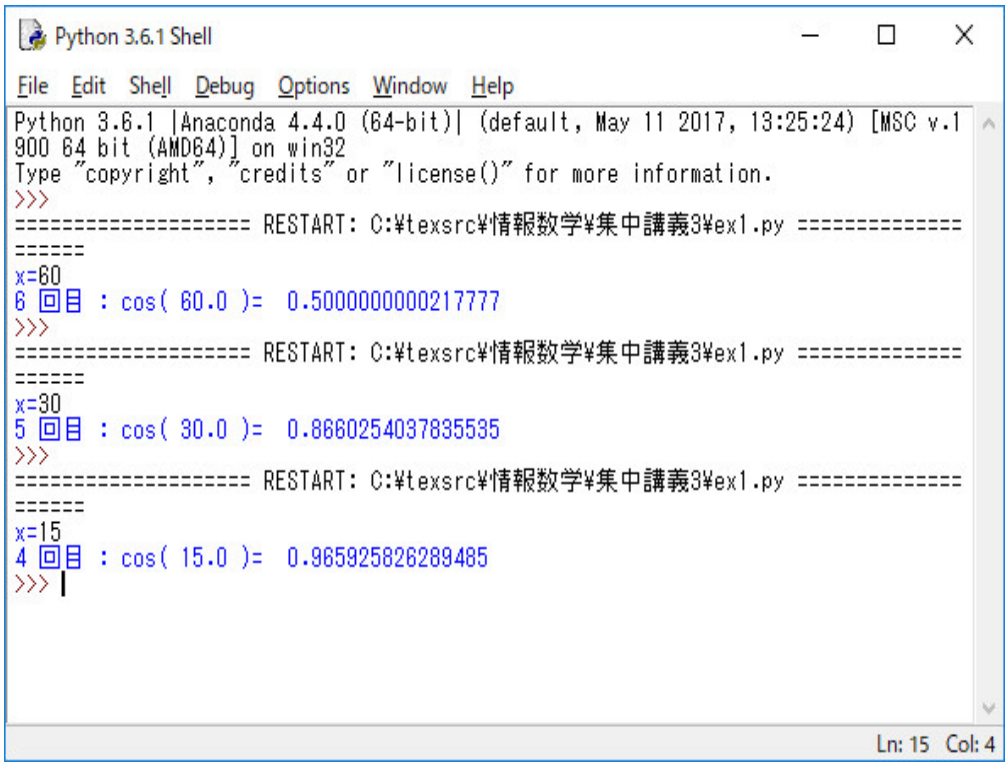
となる $cosN(N,x)$ を答えとすれば良いです。

これをそのままプログラムにすると

```
import math

def fact(n):
    r = 1.0
```

```
    for i in range(1, n+1):
        r = r * i
    return r
def cosN(n, x):
    s = 0.0
    for i in range(n+1):
        if i % 2 == 0:
            s = s + math.pow(x, 2*i) / fact(2*i)
        else :
            s = s - math.pow(x, 2*i) / fact(2*i)
    return s
x = float(input('x='))
y = x * math.pi / 180
n = 0
while True :
    n = n + 1
    if math.fabs(cosN(n, y) - cosN(n-1, y)) < 0.00000001: break
print( n,  "回目 : cos(", x, ")= ", cosN(n, y) )
```

となります。実行する。




```
while True :
    n = n + 1
    if math.fabs(cosN(n, y) - cosN(n-1, y)) < 0.00000001: break
```

は、Python には C++ のような do while 文がないのでその代用としてこのようにしています。if 文が成立して break する（while loop を抜け出す）までインデントされた部分を繰り返します。

C なら次のように表現します。

```
do {
    N++;
} while (fabs(cosN(N, y) - cosN(N-1, y)) > 0.0000001);
```

do while 文は、

```
   do {
        命令の並び
} while (条件式);
```

の形式で、while(条件式) で与えられる条件式が真 (true) である間は { と } の間の命令達を繰り返します。この場合、少なくとも一回は { と } の間の命令達を実行します。fabs() は絶対値を計算する組み込み関数（VC++ で前もって定義されている数学関数）です。do while 文の前で、

```
int N = 0;
```

と、N の宣言とN の値を 0 に初期設定しています。そして、

```
    N++;
```

で、N を一つ増やし 、

```
} while (fabs(cosN(N, y) - cosN(N-1, y)) > 0.0000001);
```

で、 $cosN(N, y)$ と $cosN(N-1, y)$ の差の絶対値が 0.0000001 より大きいかどうかを調べています。$cosN(N, y)$ と $cosN(N-1, y)$ の差の絶対値が 0.0000001 より大きければ、元に戻って、N を一つ増やし 、 $cosN(N, y)$ と $cosN(N-1, y)$ の差の絶対値が 0.0000001 より大きいかどうかを調べます。これを $cosN(N, y)$ と $cosN(N-1, y)$ の差の絶対値が 0.0000001 以下になるまで繰り返します。

これで問題は解決した訳ですが、実は問題があります。それぞれ $cosN(n, y)$ を二度づつ計算しています。$cosN(n, y)$ を計算するのに時間がかかる場合は、このような無駄は避けるように普通はします。

このためには、例えば、

```
import math

def fact(n):
    r = 1.0
    for i in range(1, n+1):
        r = r * i
    return r
def cosN(n, x):
    s = 0.0
    for i in range(n+1):
```

```
        if i % 2 == 0:
            s = s + math.pow(x, 2*i) / fact(2*i)
        else :
            s = s - math.pow(x, 2*i) / fact(2*i)
    return s
x = float(input('x='))
y = x * math.pi / 180
n = 0
new_cosN = 1.0
while True :
    print(  n,  "回目 : ", new_cosN )
    n = n + 1
    old_cosN = new_cosN
    new_cosN = cosN(n, y)
    if math.fabs(new_cosN - old_cosN) < 0.00000001: break
print( n,  "回目 : cos(", x, ")= ", new_cosN )
```

とプログラミングできます。
更に、

```
def cosN(n, x) :
    s = 0.0
    t = 1.0
    for i in range(n+1):
        if i % 2 == 0 :
            s = s + t
        else:
            s = s - t
        t = t * x * x / (2*i+1) / (2*i+2)
    return s
```

と定義すれば、

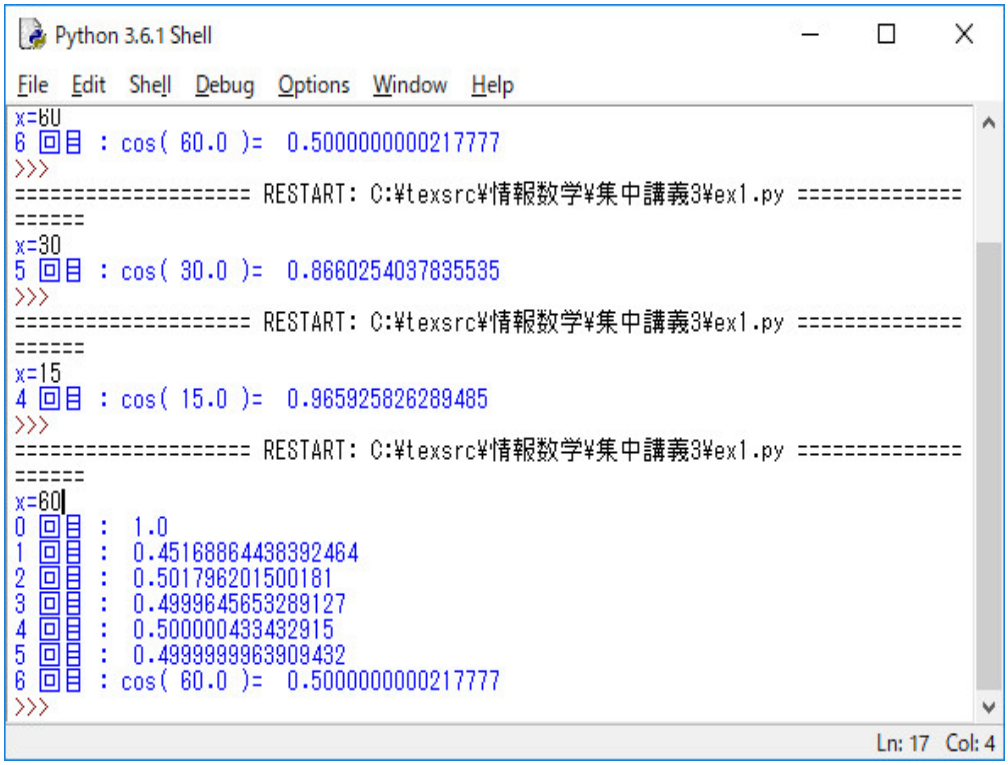
```
def fact(n):
    r = 1.0
    for i in range(1, n+1):
        r = r * i
    return r
```

の定義が不要になり、効率が良くなります。プログラム全体は

```
import math

def cosN(n, x) :
    s = 0.0
```

```
    t = 1.0
    for i in range(n+1):
        if i % 2 == 0 :
            s = s + t
        else:
            s = s - t
        t = t * x * x / (2*i+1) / (2*i+2)
    return s
x = float(input('x='))
y = x * math.pi / 180
n = 0
new_cosN = 1.0
while True :
    print(  n,  "回目 : ", new_cosN )
    n = n + 1
    old_cosN = new_cosN
    new_cosN = cosN(n, y)
    if math.fabs(new_cosN - old_cosN) < 0.00000001: break
print( n,  "回目 : cos(", x, ")= ", new_cosN )
```

となります。実行する。



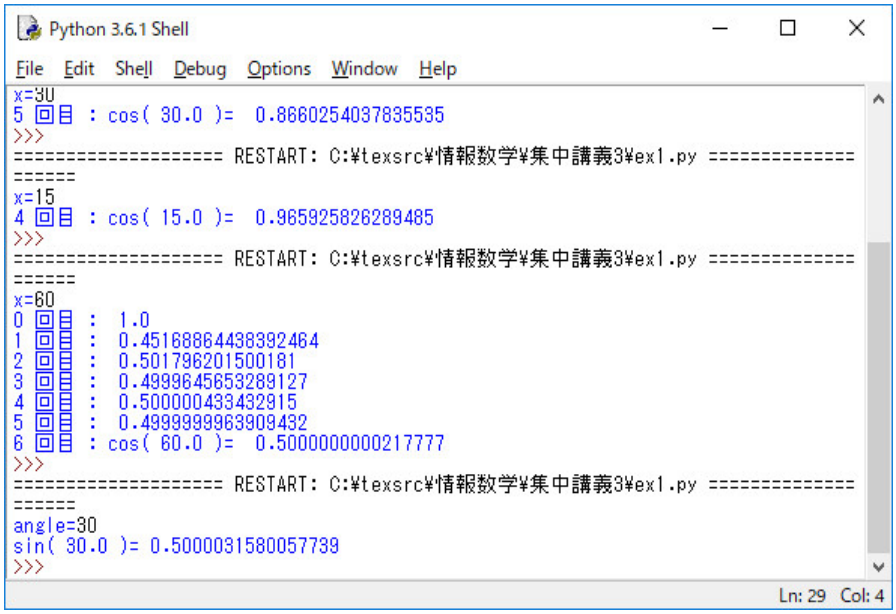
コンピュータの性能が低かった昔は色々工夫してプログラミングしていました。

また、Sheme で書いたように、x をラジアンであらわしたとき、sin(x) は x が十分小さければ x と等しいということと $\sin(x) = 3\sin(x/3) - 4\sin^3(x/3)$ という公式を用いて次のように sin(x)

の近似値を計算するプログラムを定義することが出来ます。

```
import math
def p(x):
    return 3 * x - 4 * x**3
def sine(angle):
    if abs(angle) < 0.01:
        return angle
    return p(sine(angle/3.0))
angle = float(input('angle='))
print( 'sin(', angle, ')=', sine(angle/180.0*math.pi) )
```

実行すると



となります。

演習問題

1. 整数を入力しその約数をすべて表示するプログラムを作れ。
2. $\cos(x)$ を計算するプログラムを真似て、 $\sin(x)$ を計算するプログラムを作れ。
3. $\sin(x)$ を計算するプログラムを真似て、 $\cos(x)$ を計算するプログラムを作れ。
4. ライプニッツの級数を利用して、円周率を計算するプログラムを作れ。

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots$$

がライプニッツの級数です。本当はグレゴリーの級数という方が正しいそうで、歴史的には余りにも有名ですが、円周率の計算には向かない。

5. 自然数 n について、入力された n の各桁の数字の和を求めるプログラムを作れ。

6. $a^2+b^2=c^2$ となる自然数 a, b, c の組みとその個数を求めるプログラムを作れ。但し $c<100$ とする。

7. 入力された自然数 a, b の公約数の総和を求めるプログラムを作れ。

2.1 リスト

いままで扱った変数には一つの数値しか記憶できない。多数の数値をまとめて効率よく扱うには C++ では配列を使いますが Python や Prolog や Scheme ではリストを使います。次はフィボナッチ数列の値を求めるリストを用いたプログラムである。

```
f = []
f.append(1)
f.append(1)
for i in range(2, 21):
    f.append(f[i-1]+f[i-2])
for i in range(len(f)):
    print( 'f[', i, ']=', f[i] )
```

```
f = []
```

で、変数 f に空のリストをセットしています。リストは要素を [と] で囲んだものです。要素にリストがあっても良いですし、C++ の配列と異なり、色々の要素が混ざっていても良いです。

```
f.append(1)
f.append(1)
```

で、f に 1 を追加しています。L をリストとする時、L.append(X) でリスト L の最後尾に要素 X を追加します。リストに適用できる操作には

L.append(X)	リスト L の最後尾に要素 X を追加する
L.index(X)	要素 X を探して位置を返す
L.insert(i, X)	i 番目に要素 X を挿入する
L.pop()	最後尾から要素を削除する
L.remove(X)	要素 X を削除する
`del L[i]`	i 番目の要素を削除する
L(s:e) = List	スライスへの代入

があります。

```
for i in range(2, 21):
    f.append(f[i-1]+f[i-2])
```

で、$i=2,3,4,\cdots,20$ に対して、f の最後尾に `f[i-1]+f[i-2]` を追加しています。`f[i-1]` はリスト f の $i-1$ 番目の要素です。リストの要素は 0 番目から数えます。リスト L の先頭の要素が 0 番目で、最後尾の要素が len(L)-1 番目です。この様に i 番目の要素にアクセスするだけでなく、次のようなスライス操作が出来ます。

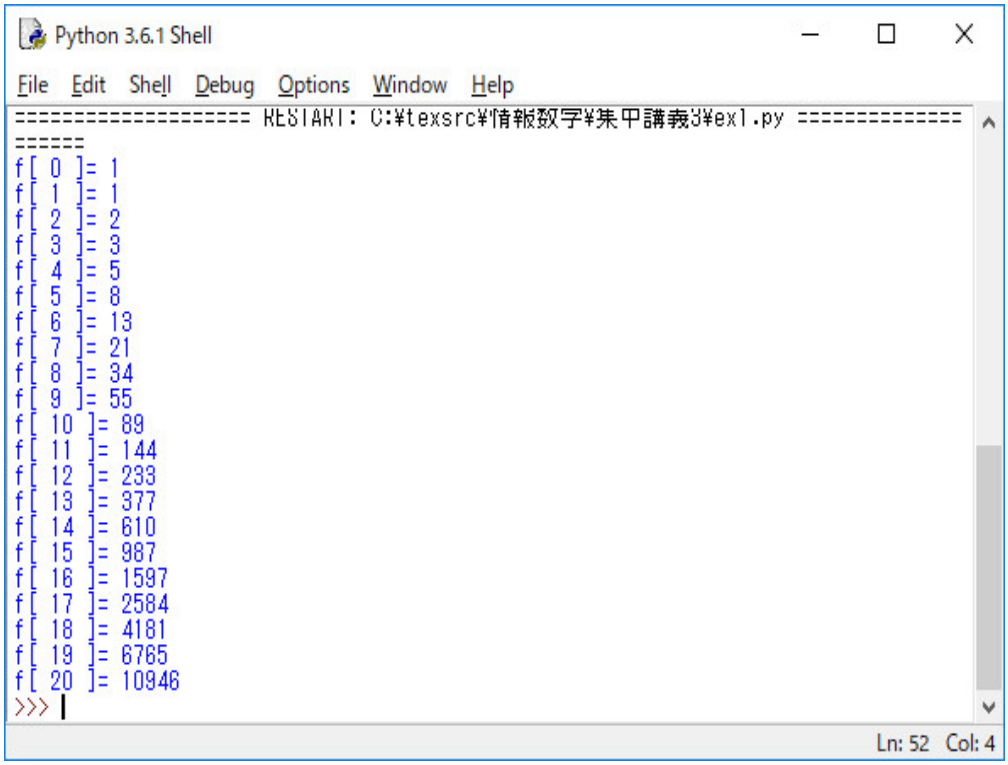
`L[start : end]`	start から end-1 まで
`L[start : ]`	start から最後尾まで
`L[ : end]`	先頭から end-1 まで
`L[ : ]`	先頭から最後尾まで

```
f = []
f.append(1)
f.append(1)
for i in range(2, 21):
    f.append(f[i-1]+f[i-2])
```

で、$f[0]=1, f[1]=1, f[2]=2, f[3]=3, f[4]=5, \cdots$ と $f[20]$ までセットされます。

```
for i in range(len(f)):
    print( 'f[', i, ']=', f[i] )
```

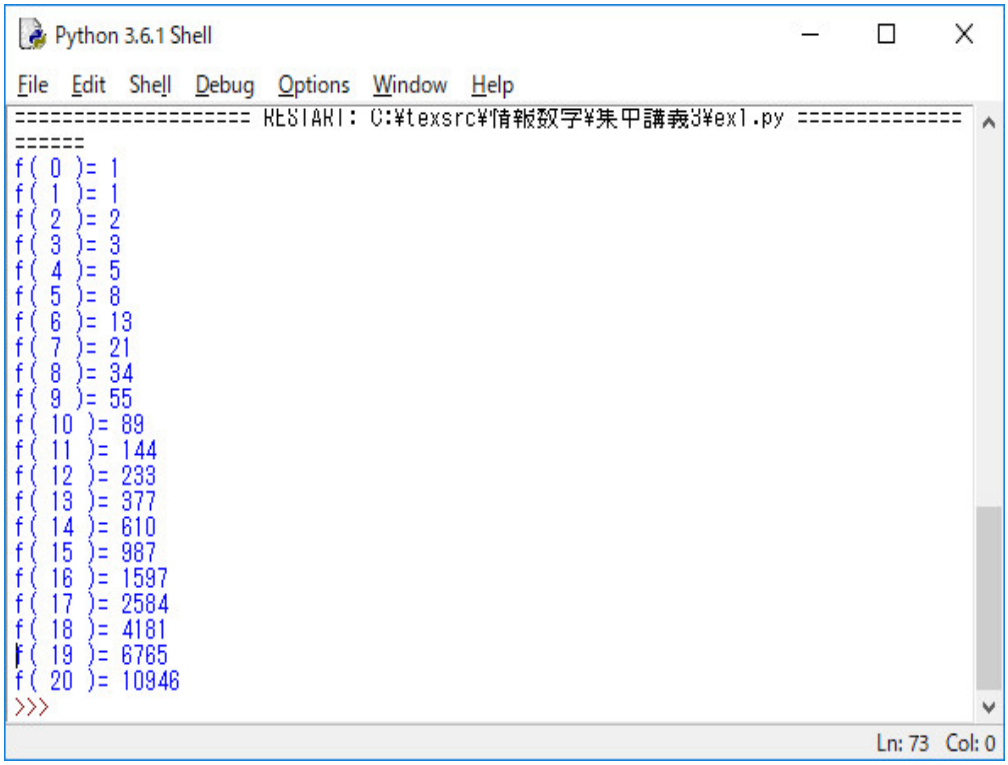
で、リスト f の全ての要素を表示しています。
実行する。



これを関数を使って、

```
def fib(n):
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n-1)+fib(n-2)
for n in range(21):
    print( "f(", n, ")=", fib(n) )
```

とすることも出来ます。実行すると



です。このように数字が小さいと問題はありませんが、大きな数字のフィボナッチ数列の値を計算する場合は、このような再起計算をさせると無駄な繰り返しが多く計算するのに時間がかかります。

世の中には Haskell という面白い関数型プログラミング言語があって、

```
fib = 1:1:zipWith (+) fib (tail fib)
```

というプログラムを fib.hs という名前で保存し、コマンドプロンプトで Haskell の対話型インタプリタ ghci を実行し、

```
:l fib.hs
```

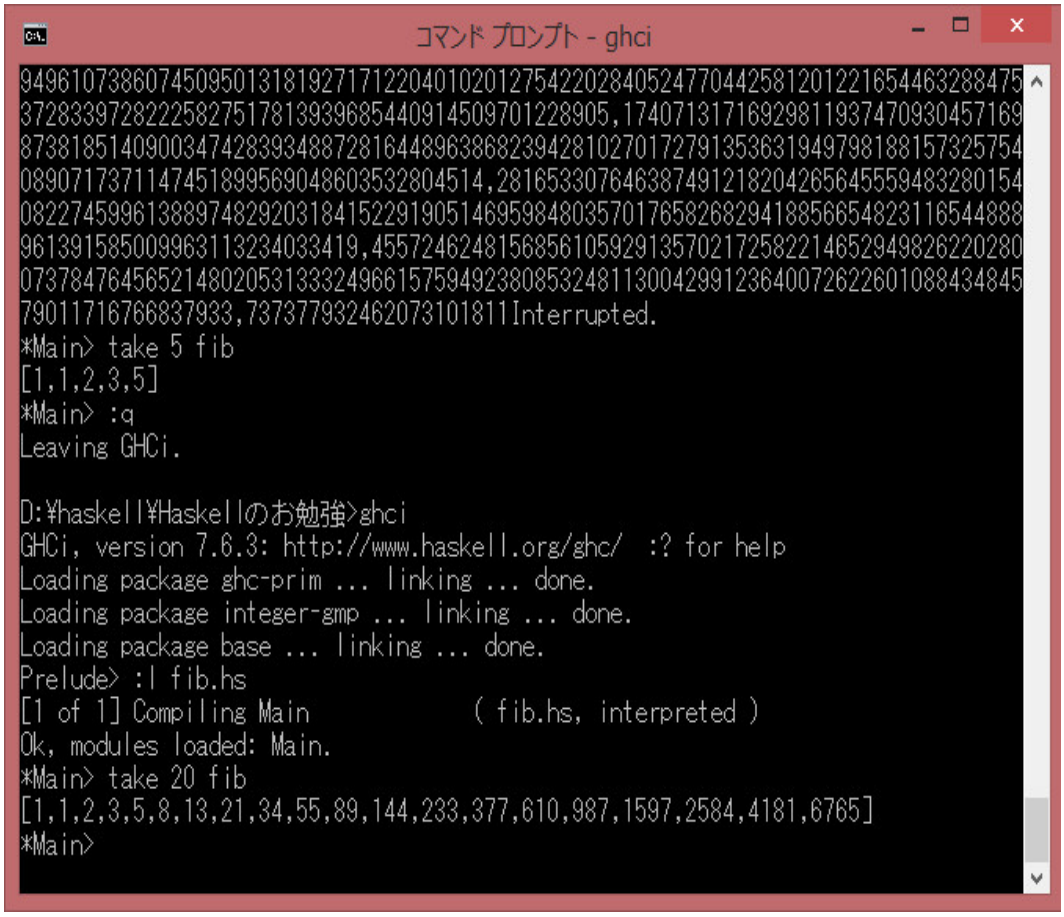
で、プログラムをロードし、

```
take 20 fib
```

とすると

```
[1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765]
```

と表示します。

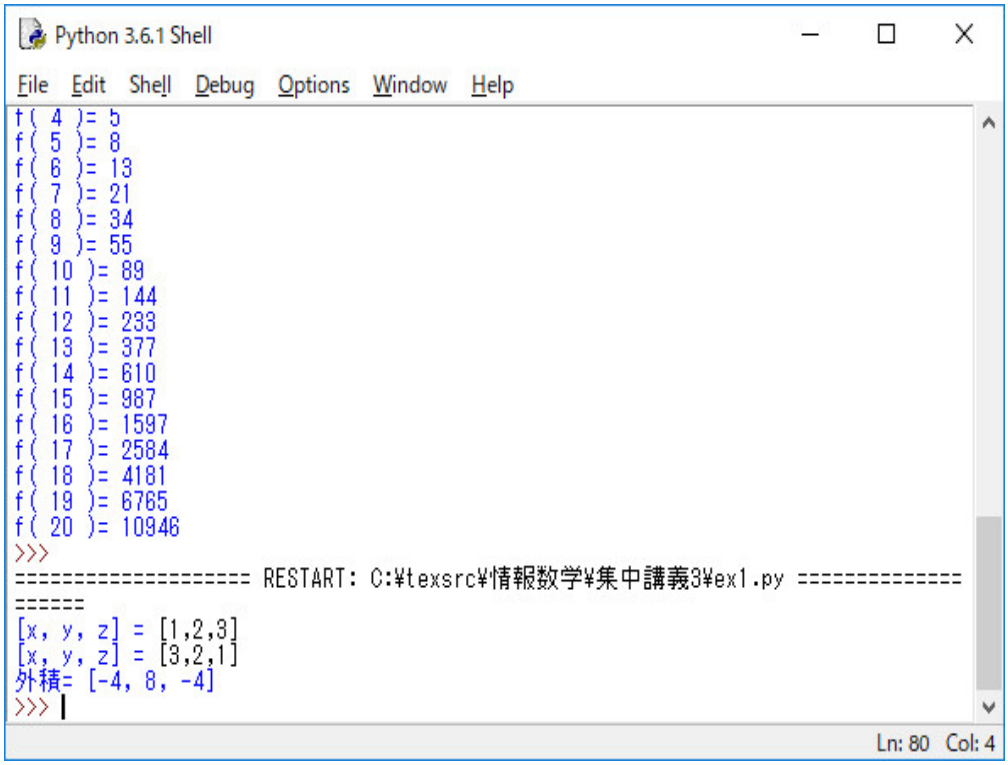


です。スペイン語は神と語るのにふさわしく、フランス語は友人と、ドイツ語は敵と、イタリア語は女性と語るのにふさわしいように、プログラミング言語にはそれぞれ得意分野があり、場合に応じて使い分けていかなければなりません。

次はベクトルの外積を計算するプログラムである。

```
def vp(a, b):
    c = []
    c.append( a[1] * b[2] - a[2] * b[1])
    c.append( a[2] * b[0] - a[0] * b[2])
    c.append( a[0] * b[1] - a[1] * b[0])
    return c
a = eval(input('[x, y, z] = '))
b = eval(input('[x, y, z] = '))
c = vp(a, b)
print( "外積=", c )
```

実行する。



ここでは、リストをキーボードから入力するのに

```
a = eval(input('[x, y, z] = '))
b = eval(input('[x, y, z] = '))
```

としています。このような方法もあります。

旺文社数学Aには次のプログラムがある。

```
10 REM 素数の発見
20 INPUT "求める素数の個数";X
30 DIM P(X)
40 J=1:P(J)=2
50 PRINT J,P(J)
60 N=3
70 FOR M=1 TO J
80    IF INT(N/P(M))*P(M)=N THEN 130
90 NEXT M
100 J=J+1
110 P(J)=N
120 PRINT J,P(J)
130 N=N+1
140 IF J<X THEN 70
150 END
```

30 行目の DIM P(X) が配列の宣言で P(0) から P(X) までの X+1 個の実数型の配列を動的に宣言している。この BASIC のプログラムは GOTO 文があるので理解するのが難しいが

1. 求める素数の個数 X を入力する。
2. 大きさ X の配列 P(X) を確保する。BASIC の配列は P(1), P(2), ···· P(X) です
3. J = 1 とし、P(J) = 2 とする。(最初の素数 2 を P(1) に保存する)
4. J と P(J) を表示する
5. N = 3 とする。次の素数の候補です。
6. すでに発見された素数で N が割り切れるか調べ割り切れればステップ 9 へ
7. N は素数なので J を一つ増やし、P(J) に保存する
8. J と P(J) を表示する
9. N を一つ増やす
10. 素数が X 個見つかっていなければ、ステップ 6 へ

ということをしています。Python 用のもっと分かりやすいアルゴリズムは

1. 求める素数の個数 X を入力する。
2. 空の リスト P を作る。
3. P の最後尾に 2 を追加する。(最初の素数 2 を P[0] に保存する)
4. len(P) と P[len(P)-1] を表示する
5. N = 3 とする。次の素数の候補です。
6. 素数が X 個見つかるまで次を繰り返す。即ち、len(P) == X になるまでと言うこと
   (a) 合成数であったかどうかを調べるための変数 FLAG を眞にセット
   (b) すでに発見された素数で N が割り切れるか順に調べ割り切れれば FLAG に偽をセット
   (c) FLAG が眞であれば N は素数なので P の最後尾に N を保存し、len(P) と P[len(P)-1] を表示する
   (d) N を一つ増やす

求める素数の個数 X を入力する。
をプログラミングします。

```
X = int(input( "求める素数の個数："))
```

となります。
空の リスト P を作る。

```
P = []
```

とします。これを組み込みます。

```
X = int(input( "求める素数の個数："))
P = []
```

P の最後尾に 2 を追加する。(最初の素数 2 を P[0] に保存する) は

```
P.append(2)
```

で良いです。これを組み込みます。

```
X = int(input( "求める素数の個数："))
P = []
P.append(2)
```

len(P) と P[len(P)-1] を表示するは

```
print( len(P),  " : ", P[len(P)-1] )
```

で良いです。見つかった素数の個数は len(P) で、最後に見つけた素数は `P[len(P)-1]` です。これを組み込みます。

```
X = int(input( "求める素数の個数："))
P = []
P.append(2)
print( len(P),  " : ", P[len(P)-1] )
```

N = 3 とする。次の素数の候補です。は

```
N = 3
```

で良いです。これを組み込みます。

```
X = int(input( "求める素数の個数："))
P = []
P.append(2)
print( len(P),  " : ", P[len(P)-1] )
N = 3
```

素数が X 個見つかるまで次を繰り返す。は while 文を使って

```
while len(P) < X :
```

で良いです。これを組み込みます。

```
X = int(input( "求める素数の個数："))
P = []
P.append(2)
print( len(P),  " : ", P[len(P)-1] )
N = 3
while len(P) < X :
```

合成数であったかどうかを調べるための変数 FLAG を眞にセットは bool の値を保持する変数 flag を使って

```
    flag = True
```

で良いです。これを組み込みます。

```
X = int(input( "求める素数の個数："))
P = []
P.append(2)
print( len(P),  " : ", P[len(P)-1] )
N = 3
while len(P) < X :
    flag = True
```

すでに発見された素数で N が割り切れるか順に調べ割り切れれば FLAG に偽をセットは for 文を使って

```
    for i in P:
        if N % i == 0 :
            flag = False
            break
```

で良いです。これを組み込みます。

```
X = int(input( "求める素数の個数："))
P = []
P.append(2)
print( len(P),  " : ", P[len(P)-1] )
N = 3
while len(P) < X :
    flag = True
    for i in P:
        if N % i == 0 :
            flag = False
            break
```

FLAG が眞であれば N は素数なので P の最後尾に N を保存し、len(P) と P[len(P)-1] を表示するは if 文を使って

```
    if flag :
        P.append(N)
        print( len(P),  " : ", P[len(P)-1] )
```

で良いです。これを組み込みます。

```
X = int(input( "求める素数の個数："))
P = []
```

```
P.append(2)
print( len(P),  " : ", P[len(P)-1] )
N = 3
while len(P) < X :
    flag = True
    for i in P:
        if N % i == 0 :
            flag = False
            break    if flag :
        P.append(N)
        print( len(P),  " : ", P[len(P)-1] )
```

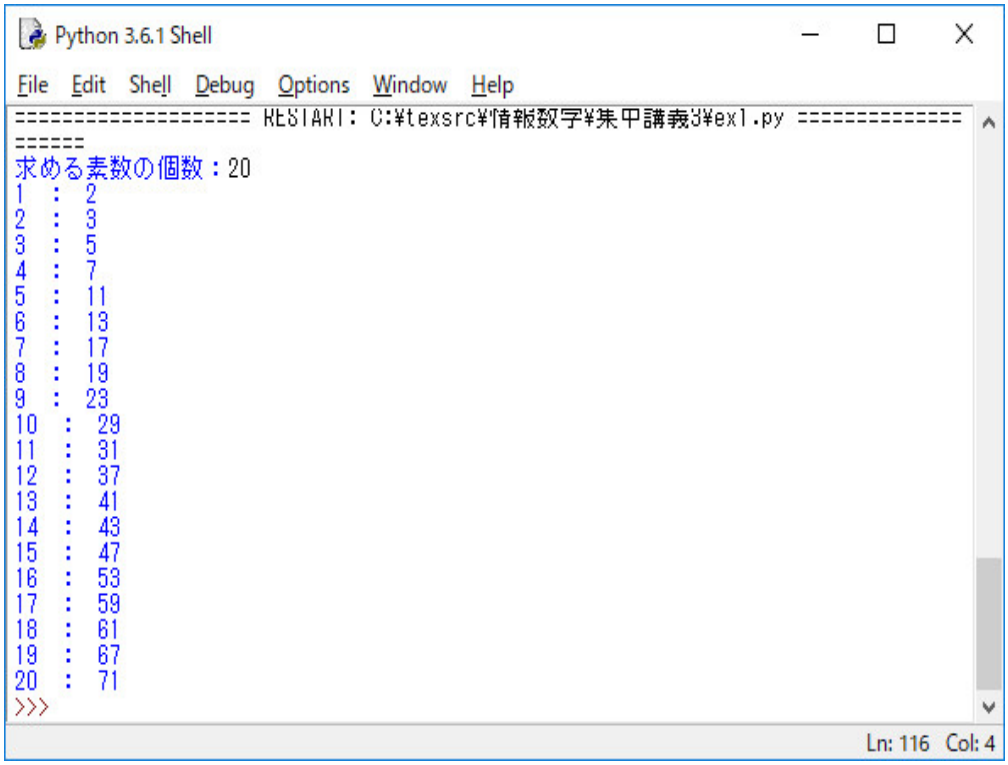
N を一つ増やすは

```
    N = N + 1
```

で良いです。これを組み込みます。

```
X = int(input( "求める素数の個数："))
P = []
P.append(2)
print( len(P),  " : ", P[len(P)-1] )
N = 3
while len(P) < X :
    flag = True
    for i in P:
        if N % i == 0 :
            flag = False
            break
    if flag :
        P.append(N)
        print( len(P),  " : ", P[len(P)-1] )
    N = N + 1
```

完成しました。実行する。



例題：配列の要素を並べ替える（ソートと言います）プログラムを作れ。次は入力した数値を大きい順に並べ替える BASIC のプログラムである。

```
10 REM データの並べ替え
20 INPUT "データの個数";N
30 DIM A(N)
40 FOR J=1 TO N
50     INPUT A(J)
60 NEXT J
70 FOR J=1 TO N-1
80     FOR K=J+1 TO N
90         IF A(J) > A(K) THEN 110
100        X=A(J):A(J)=A(K):A(K)=X
110    NEXT K
120    PRINT A(J)
130 NEXT J
140 PRINT A(N)
150 END
```

これは最小値選択法と呼ばれるアルゴリズムです。データの個数が多くなると色々なアルゴリズムを使う必要があります。この分野は最も研究されている分野の 1 つです。

これをリストを使って Python に翻訳すると次のようになる。

```
n = int(input("データの個数= "))
a = []
```

```
for j in range(n):
    s = str(j+1)+ '= '
    a.append(float(input(s)))
for j in range(n-1) :
    for k in range(j+1, n) :
        if a[j] > a[k] : continue
        x = a[j]
        a[j] = a[k]
        a[k] = x
    print (a[j] )
print( a[n-1] )
```

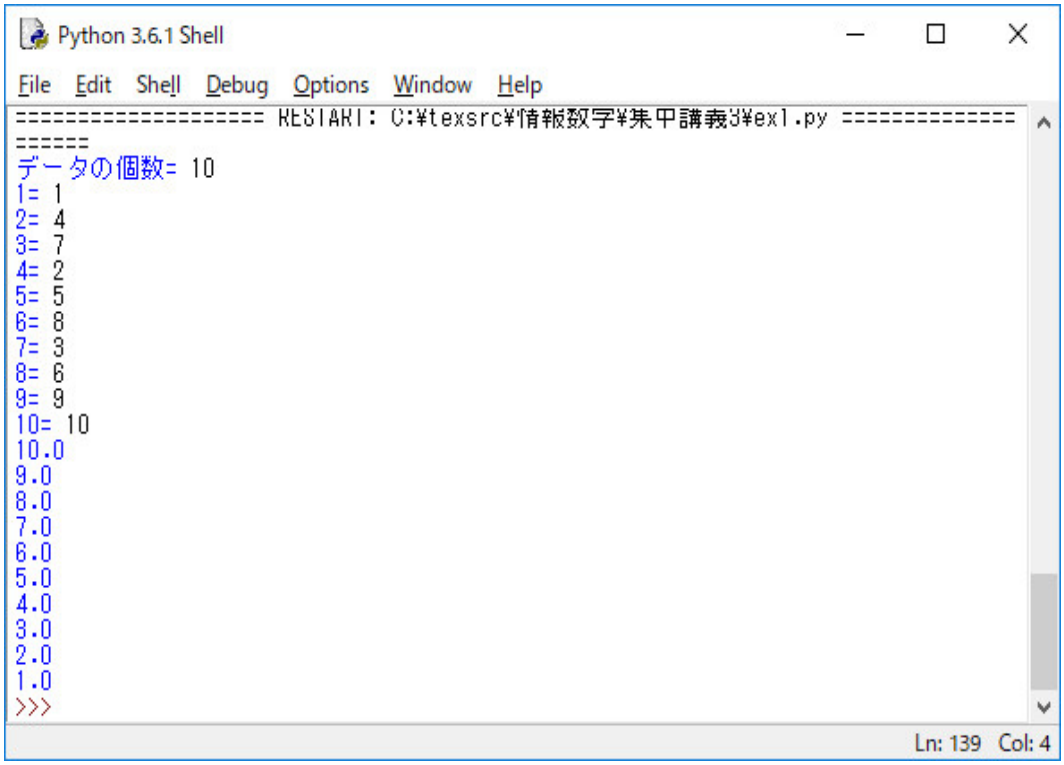
Python にはリストのサイズを初期設定する命令は無いので（自分でダミーの要素を持ったリストを作れば良いですが)、空のリストからデータが入力されるたびに append で追加しています。

```
    s = str(j+1)+ '= '
```

は、整数値 j+1 を文字列に変換し、 '= ' を連接しています。
continue 文が使われていることに注意。

```
    for k in range(j+1, n) :
        if a[j] > a[k] : continue
        x = a[j]
        a[j] = a[k]
        a[k] = x
```

で、if 文で $a[j] > a[k]$ であれば、以下の命令達を無視し、すぐ k を一つ増やして for 文 を実行します。実行します。



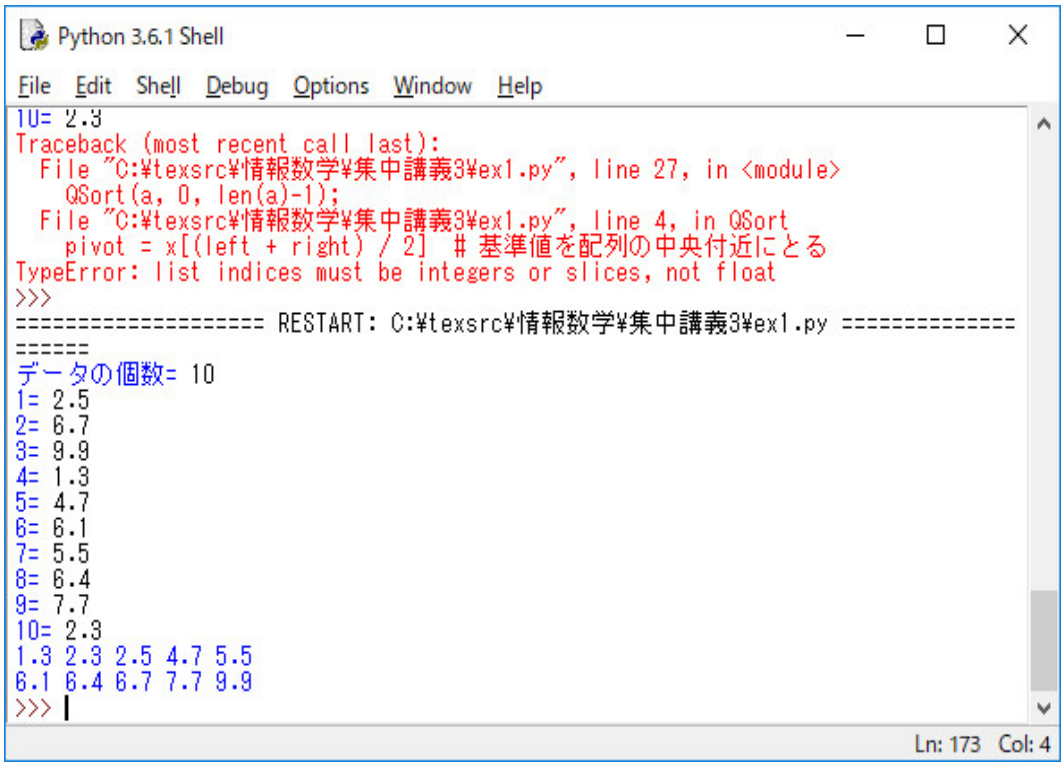
普通はすべて並べ替えてから整列したデータを表示します。

ランダムなデータなら高速でソートできるアルゴリズムにクイックソートというのがあります。大量のデータをソートするときに威力を発揮します。クイックソートのプログラムを作ってみよう。

```
def  QSort(x, left, right) :
    i = left                        # ソートする配列の一番小さい要素の添字
    j = right                       # ソートする配列の一番大きい要素の添字
    pivot = x[(left + right) // 2]  # 基準値を配列の中央付近にとる

    while True :                    # 無限ループ
        while x[i] < pivot :        # pivot より大きい値が
            i += 1                  # 出るまで i を増加させる
        while pivot < x[j] :        # pivot より小さい値が
            j -= 1                  #  出るまで j を減少させる
        if  i >= j :                # i >= j なら
            break                   # 無限ループから抜ける
        temp = x[i]                 # x[i] と x[j] を交換
        x[i] = x[j]
        x[j] = temp
        i += 1                      # 次のデータ
        j -= 1
    if  left < i - 1:               # 基準値の左に 2 以上要素があれば
        QSort(x, left, i - 1)       # 左の配列をクイックソートする
    if  j + 1 <  right :            # 基準値の右に 2 以上要素があれば
```

```
        QSort(x, j + 1, right)      # 右の配列をクイックソートする
n = int(input("データの個数= "))
a = []
for j in range(n):
    s = str(j+1)+ '= '
    a.append(float(input(s)))
QSort(a, 0, len(a)-1);
for i in range(n):
    print( a[i] , end=' ')
    if (i+1) % 5 == 0 :
        print()
```

実行します。



上のエラーは

```
    pivot = x[(left + right) / 2]  # 基準値を配列の中央付近にとる
```

としていたからです。これでは (left + right) / 2 が不動点少数値になります。整数値にするには

```
    pivot = x[(left + right) // 2]  # 基準値を配列の中央付近にとる
```

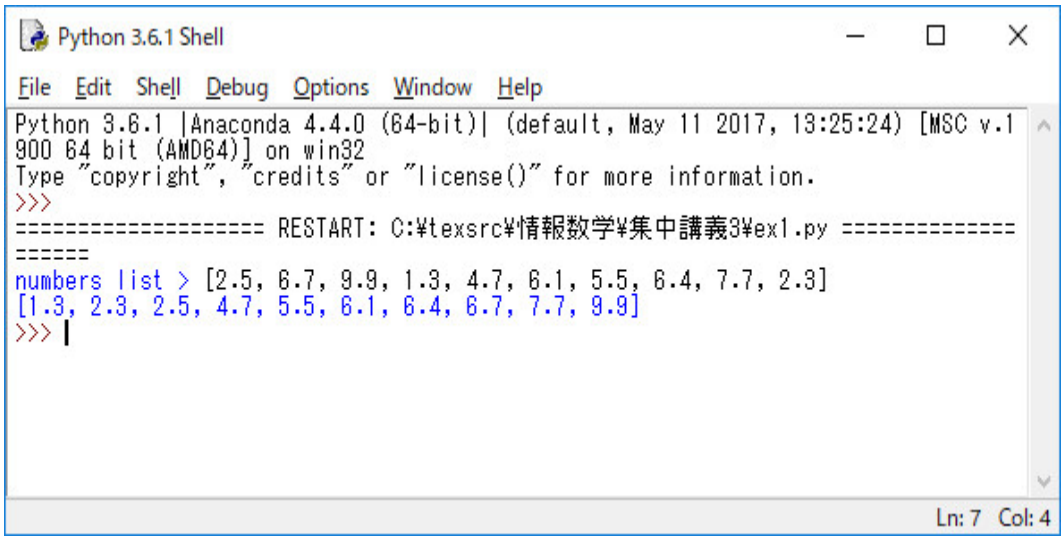
とする必要がありました。

これらのプログラム（インサートソートとクイックソートのプログラム）を **Prolog** や **Scheme** のインサートソートとクイックソートのプログラムと見比べて下さい。やっていることは同じですが、そのアルゴリズムの表現方法が言語によって違います。

実際は Python のリストには sort() という関数が定義されていて

```
a = eval(input('numbers list > '))
a.sort()
print( a )
```

を実行すると



となります。

更に、Haskell という数学者ご愛用の関数型プログラミング言語があり、これではクイックソートのプログラムを

```
module Qsort where

qsort _ [] = []
qsort f (x:xs) = before ++ (x : after)
 where before = qsort f $ filter (not . (f x)) xs
       after  = qsort f $ filter (f x) xs
```

や

```
qsort _ []  = []
qsort f (x : xs) = qsort f before ++ [x] ++ qsort f after
                 where
                   before = [a| a <- xs, (f x) a]
                   after  = [b | b <- xs, not $ (f x) b]
```

で書くことができます。これを Qsort.hs という名前で保存し、コマンドプロンプトで ghci を起動し、

```
:l Qsort.hs
```

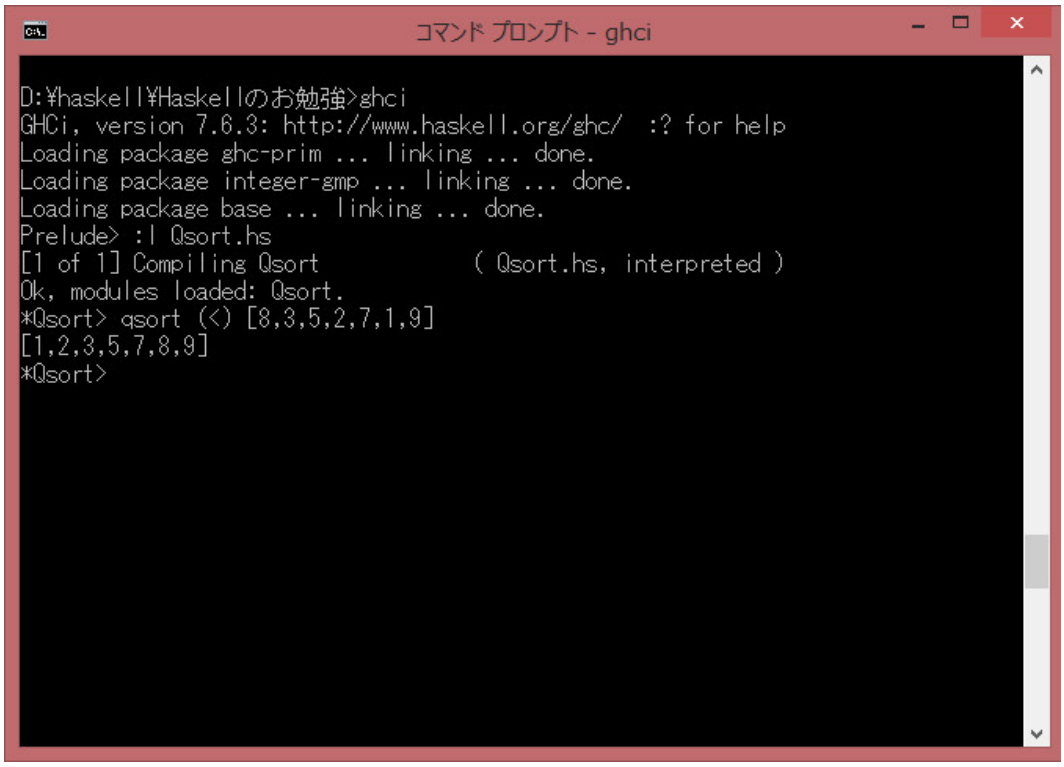
で、Qsort.hs をロードし、

```
qsort  (<) [8,3,5,2,7,1,9]
```

を実行すれば

```
[1,2,3,5,7,8,9]
```

となります。



このプログラムには”どういう値が欲しいか?”を記述しているだけで、その値をどういう手順で求めるかはほとんど気にする必要はありません。Haskell は特に数値計算に有効なプログラミング言語です。

ソートの色々なアルゴリズムやその性能を知りたければ、図書館で Knuth の「The Art of Computer Programming」を探して読んでみると良いです。シリーズの一冊の半分を使って、ソートのアルゴリズムを解説しています。この本は現在でも手に入ります。コンピュータ・サイエンスのバイブルで本箱に飾っておくにはいい本ですが、実際に全部通して読んだ人はあまりいないと思います。辞書のように使います。もっと読みやすい本の中では THOMAS H. CORMEN 他著「INTRODUCTION TO ALGORITHMS」 The MIT Press が現在は評価が高いみたいです。これも厚い本ですが、本格的にプログラミングを勉強したくなったら気に入った部分を読んでみるといいです。

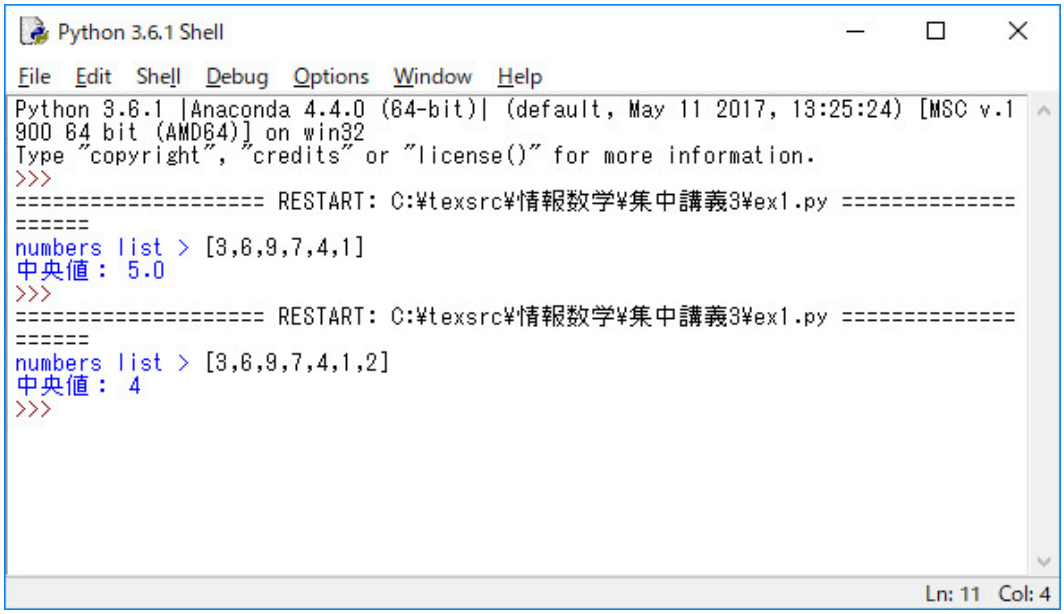
例題：四分位数を求めるプログラムを作れ。高校の数学では、中央値、四分位数、箱ひげ図を学びます。ここでは、ソートのプログラムを説明したので、中央値、四分位数を計算するプログラムを作ってみます。中央値はデータの個数が奇数の時は、小さい順にソートした時、中央にある値です。偶数の時は、中央２個の値の平均です。第２四分位数は中央値のことで、第１四分位数は中央値を除いた下位のデータの中央値で、第３四分位数は中央値を除いた上位のデータの中央値です。まず、中央値を求めます。ソートは Python に組み込まれている関数 sort() を使います。

```
a = eval(input('numbers list > '))
```

```
a.sort()
if len(a) % 2 == 1 :
    print( "中央値：", a[len(a)//2] )
else:
    print( "中央値：", (a[len(a)//2-1]+a[len(a)//2])/2 )
```

で良いです。

プログラムを実行すると



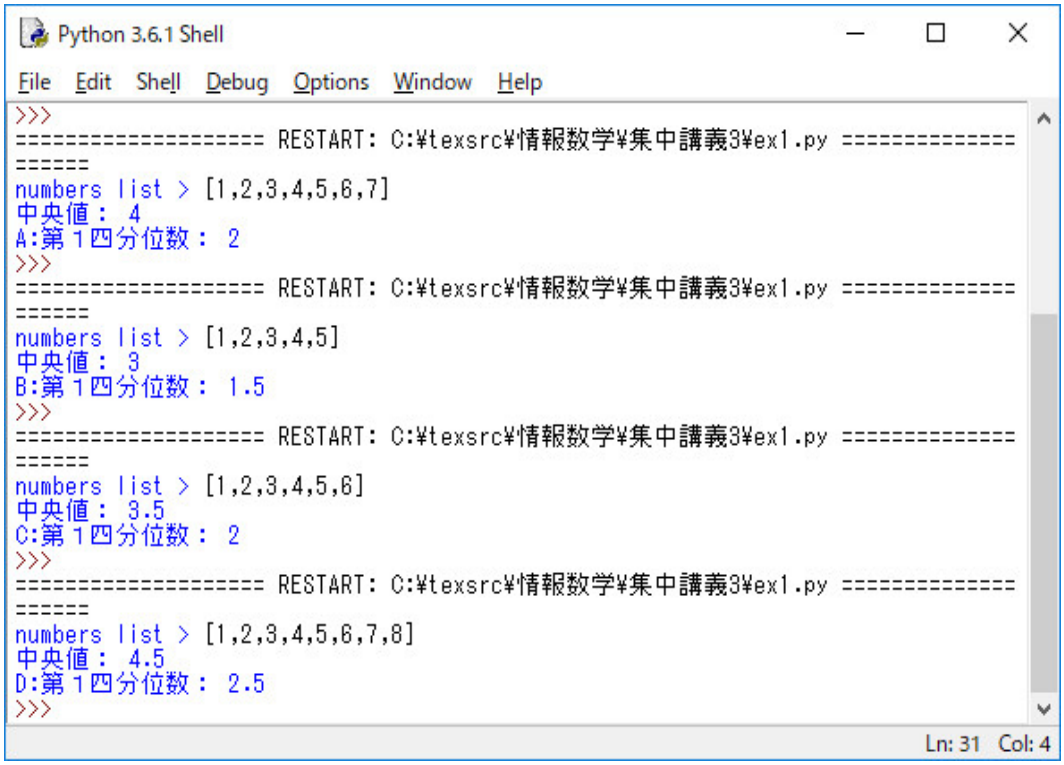
です。図は Python2.7 になっていますが、上のプログラムは Python3.6 のプログラムです。図を作り直すのが面倒ですから、昔の図のままです。以下、同様です。

次に、第 1 四分位数（中央値を除いた下位のデータの中央値）を求めてみます。難しいことはありません。定義通り場合分けすればいいです。

```
a = eval(input('numbers list > '))
a.sort()
if len(a) % 2 == 1 :
    print( "中央値：", a[len(a)//2] )
else:
    print( "中央値：", (a[len(a)//2-1]+a[len(a)//2])/2.0 )
if len(a) % 2 == 1:
    if (len(a)//2) % 2 == 1:
        print( "A:第 1 四分位数：", a[(len(a)//2)//2] )
    else:
        print( "B:第 1 四分位数：", (a[(len(a)//2)//2-1]+a[(len(a)//2)//2])/2.0  )
else:
    if (len(a)//2) % 2 == 1:
        print( "C:第 1 四分位数：", a[(len(a)//2)//2] )
```

```
    else:
        print( "D:第１四分位数：", (a[(len(a)//2)//2-1]+a[(len(a)//2)//2])/2.0 )
```

で良いです。デバッグの為に、どこで計算した第１四分位数か分かるように記号を付けていますが、プログラムが正しいことが確認できれば、省略します。プログラムを実行すると



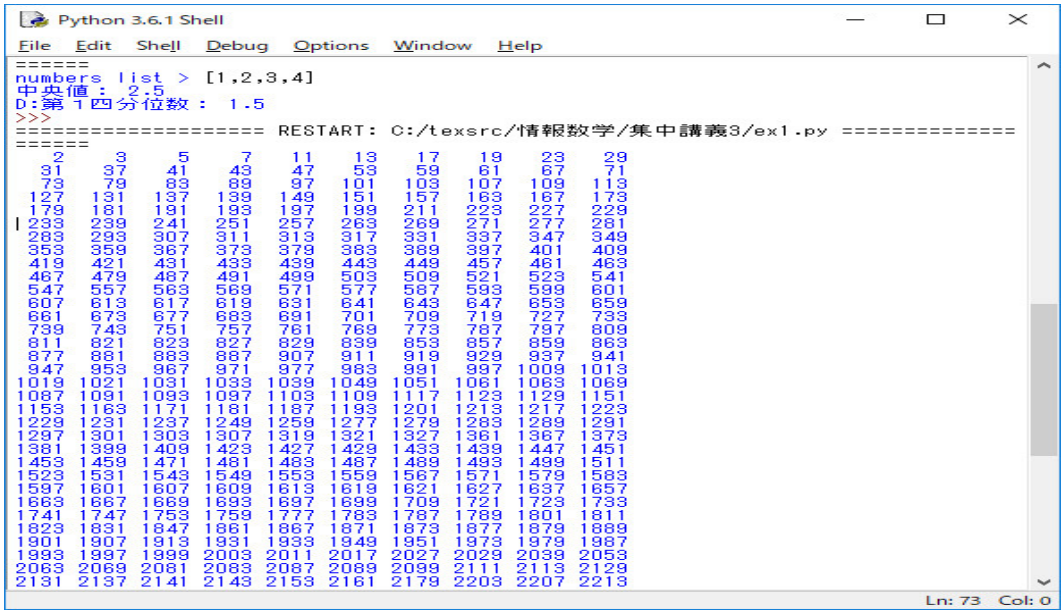
です。、第３四分位数（中央値を除いた上位のデータの中央値）を計算するプログラムと箱ひげ図を描くプログラムは演習問題とします。

例題：５０００未満の素数をすべて求めるプログラムを作れ。ここではエラトステネスの篩という有名なギリシャ時代から知られているアルゴリズムを使った例を示します。Python で作ると次のようになります。

```
import math

p = []
for i in range(5001) :
    p.append(1)
p[0] = 0
p[1] = 0
for i in range(4, 5001, 2) :
    p[i] = 0
for i in range(3, int(math.sqrt(5000))+1, 2) :
    if not p[i] : continue
    for k in range(2*i, 5001, i) :
        p[k] = 0
```

```
k = 0
for i in range(5001) :
    if p[i] :
        print( "%4d" % i , end=' ')
        k += 1
        if k % 10 == 0 :
            print()
```

実行します。



次は２次正方行列を入力し、その行列式を計算するプログラムである。

```
def det(a) :
    return a[0][0]*a[1][1]-a[1][0]*a[0][1]
a = eval(input("二次正方行列="))
print( "行列式 =", det(a))
```

実行します。

二次正方行列=[[1,2],[3,2]]
行列式 = -4

のように計算してくれます。行列は $[[1,2],[3,2]]$ のように、リストのリストで表しています。キーボードでこのようなリストのリストを読み込むには

```
a = eval(input("二次正方行列="))
```

とします。関数 eval() ですが、正確な解説をしている文献が見当たらないですが、これをかぶせることで input() 函数を Python 2.7 のように使うことが出来ます。今の場合、リストのリスト [[1,2],[3,2]] をキーボード入力すると文字列 '[[1,2],[3,2]]' ではなくリスト [[1,2],[3,2]] が a にセットされます。

```
def det(a) :
    return a[0][0]*a[1][1]-a[1][0]*a[0][1]
```

は、２次正方行列の行列式の高等学校で習う定義をそのまま計算しています。
　３次の正方行列の行列式の計算も行列式の定義をそのまま計算すればいいです。

```
def det(a) :
    d =  a[0][0]*a[1][1]*a[2][2]+a[1][0]*a[2][1]*a[0][2]
    d += a[2][0]*a[1][2]*a[0][1]-a[2][0]*a[1][1]*a[0][2]
    d -= a[0][1]*a[1][0]*a[2][2]+a[0][0]*a[2][1]*a[1][2]
    return d
a = eval(input("三次正方行列="))
print( "行列式 =", det(a))
```

実行します。

```
三次正方行列=[[0,1,1],[1,1,0],[1,1,1]]
行列式 = -1
```

のように計算してくれます。
　４次以上の正方行列の行列式の計算方法は大学の線形代数で学ぶので、その筆算のためのアルゴリズムをプログラミングすれば良いですが、複雑でちょっと厄介です。

1. 「**行列 $\mathbf{A}$ の i 行から、j 行の定数倍を引いても、行列式の値は変わらない**」という性質を使って行列 A を対角行列に変換する。
2. 行列式の定義を見ると、**「対角行列の行列式は、対角成分の積と等しい」**。（三角行列でいいですが、ここでは対角行列に変換しています。）

　行列式を計算するプログラムの模範解答は、これをプログラミングすることです。例えば、次のようになります。

```
def determinant(A):
    for i in range(len(A)):
        if A[i][i] == 0:
            k = 0
            while k < len(A):
                if A[k][i] != 0:
                    break
                k += 1
            if k == len(A):          # all numbers in a column is zero
                return 0
            for j in range(len(A)):
                A[i][j] += A[k][j]
        for j in range(len(A)):
            if i == j:
                continue
```

```
            c = float(A[j][i]) / A[i][i]
            for k in range(len(A)):
                A[j][k] -=  c * A[i][k]
    det = 1.0;
    for i in range(len(A)):
        det *= A[i][i]
    return det
A = eval(input('matrix = '))
print( determinant(A))
```

実行します。

```
matrix = [[1,2,3,4],[0,1,1,1],[1,0,2,1],[3,1,3,2]]
-6.0
```

です。行列はリストのリスト

```
 [[1,2,3,4],
  [0,1,1,1],
  [1,0,2,1],
  [3,1,3,2]]
```

で表します。行列の読込は

```
A = eval(input('matrix = '))
```

で実行します。input() 関数を eval() 関数で囲うことで目的を達することが出来ます。

```
def determinant(A):
    for i in range(len(A)):
        if A[i][i] == 0:
            k = 0
            while k < len(A):
                if A[k][i] != 0:
                    break
                k += 1
            if k == len(A):          # all numbers in a column is zero
                return 0
            for j in range(len(A)):
                A[i][j] += A[k][j]
        for j in range(len(A)):
            if i == j:
                continue
            c = float(A[j][i]) / A[i][i]
            for k in range(len(A)):
                A[j][k] -=  c * A[i][k]
```

```
    det = 1.0;
    for i in range(len(A)):
        det *= A[i][i]
    return det
```

が行列式を計算する関数 determinant(A) です。
アルゴリズムは

行列 A の各行 i に対して
A[i][i] = 0 なら
A[k][i] != 0 となる行 k を探す
なければ 行列式は 0 です。0 を返す。
あれば
i 行に k 行を加える
（これで A[i][j] != 0 になった）
行列 A の各行 j に対して
j == i なら何もしない
c = A[j][i]/A[i][i] とし
j 行から i 行の c 倍を引く
（この操作で i 列は A[i][i] 以外 0 になる）
最後にこの対角行列の対角要素の積を返す

です。
関数 determinant(A) はこのアルゴリズムを Python のコードに書き変えただけです。
実際は

A[k][i] != 0 となる行 k を探す

を

A[k][i] != 0 となる絶対値 |A[k][i]| 最大の行 k を探す

と変えた方が良いアルゴリズムとなります。
注意:関数 determinant(A) は A を変形するので、後で A を使いたい場合は関数 determinant(A) を呼ぶ前に A をコピーしておく必要があります。その場合、コピーするのに

```
old_A = A[:]
```

としたのでは、これは浅いコピーといわれるもので、リストのリストである A のようなリストではコピーが不十分です。このような場合は深いコピーと呼ばれる

```
old_A = copy.deepcopy(A)
```

のような deepcopy() 関数による処理が必要です。この為に

```
import copy
```

のインポート文が必要です。
行列式の計算は NumPy を使えば簡単です。

```
import numpy as np
A = np.array([ [0, 1, 2],
               [1, 2 ,3],
               [2, 2, 1] ])
detA = np.linalg.det(A)
print( detA )
B = np.mat("1 2 3 4; 0 1 2 1; 5 1 3 2; 2 -1 8 1")
detB = np.linalg.det(B)
print( detB )
```

のように、det() という関数を呼ぶだけです。行列は mat() 関数を使って定義することもできます。実行すると

```
1.0
96.0
```

です。

ここでは別のアルゴリズムによる行列式の計算に挑戦してみましょう。プログラミングの練習だと思ってやってみましょう。関数型プログラミング言語 Haskell によるプログラミングでこのプログラムを作ったことがありますが、結構大変でしたが、Python には色々便利な機能があるので意外と簡単です。

そのために、順列組み合わせのプログラムを作って置きましょう。順列組み合わせは高等学校で学びます。順列組み合わせの個数の計算は得意でも、すべての要素を書きだすのは不得意かもしれません。これも高級なアルゴリズムが知られて、参考書も出版されていますが、ここでも模範解答ではなく、再帰呼び出しを使った単純なアルゴリズムをプログラミングしてみます。

順列のすべての要素を書きだすプログラムは

```
l = []
r = []
def get_perm(l, s, k):
    global r
    if len(s)==k:
        r.append(s)
        return
    for e in l:
        old_s = s[:]
        s.append(e)
        new_l = l[:]
        new_l.remove(e)
        get_perm(new_l, s, k)
        s = old_s
n = int(input('n='))
k = int(input('k='))
l = [i for i in range(1, n+1)]
get_perm(l, [], k)
```

```
print(len(r))
print(r)
```

です。実行結果は

```
n=5
k=3
60
[[1, 2, 3], [1, 2, 4], [1, 2, 5], [1, 3, 2], [1, 3, 4], [1, 3, 5],
[1, 4, 2], [1, 4, 3], [1, 4, 5], [1, 5, 2], [1, 5, 3], [1, 5, 4],
 [2, 1, 3], [2, 1, 4], [2, 1, 5], [2, 3, 1], [2, 3, 4], [2, 3, 5],
 [2, 4, 1], [2, 4, 3], [2, 4, 5], [2, 5, 1], [2, 5, 3], [2, 5, 4],
 [3, 1, 2], [3, 1, 4], [3, 1, 5], [3, 2, 1], [3, 2, 4], [3, 2, 5],
 [3, 4, 1], [3, 4, 2], [3, 4, 5], [3, 5, 1], [3, 5, 2], [3, 5, 4],
 [4, 1, 2], [4, 1, 3], [4, 1, 5], [4, 2, 1], [4, 2, 3], [4, 2, 5],
 [4, 3, 1], [4, 3, 2], [4, 3, 5], [4, 5, 1], [4, 5, 2], [4, 5, 3],
 [5, 1, 2], [5, 1, 3], [5, 1, 4], [5, 2, 1], [5, 2, 3], [5, 2, 4],
 [5, 3, 1], [5, 3, 2], [5, 3, 4], [5, 4, 1], [5, 4, 2], [5, 4, 3]]
```

です。さて、

```
l = []
r = []
n = int(input('n='))
k = int(input('k='))
l = [i for i in range(1, n+1)]
```

の部分が準備です。l と r を空リストで初期化しています。そして、全体の要素の個数 n と並べる個数 k を入力しています。

```
l = [i for i in range(1, n+1)]
```

は、リスト内包表記と呼ばれるもので、一般形は

```
l = [追加する要素 for 取り出した要素 in リスト (if  条件式)]
```

です。 if 条件式 は省略可能です。

```
l = []
for 取り出した要素 in リスト:
   if  条件式:
       l.append(追加する要素)
```

と同じことです。

```
l = [i for i in range(1, n+1)]
```

は、単にリスト $l = [1, 2, \cdots, n]$ とセットしています。この n 個の要素からなるリストから r 個取り出し、並べ、結果のリストをリストのリストとして r に集計します。

```
def get_perm(l, s, k):
    global r
    if len(s)==k:
        r.append(s)
        return
    for e in l:
        old_s = s[:]
        s.append(e)
        new_l = l[:]
        new_l.remove(e)
        get_perm(new_l, s, k)
        s = old_s
```

が本体の再帰関数で

```
get_perm(l, [], k)
```

の形で呼び出されています。リストのリスト r は `get_perm(l, s, k)` の中でグローバル変数として宣言されています。

```
    global r
```

従って、 `get_perm(l, s, k)` の内部で r の値を変更した結果が `get_perm(l, s, k)` が終了した後でも有効です。`get_perm(l, s, k)` の引数 l は順列の並びにまだ使われていない要素を保持するリストで、初期状態はすべての要素がまだ順列の要素として洗濯されていませんから、$l = [1, 2, \cdots, n]$ です。引数 s には順列の並びとして採用した要素がセットされます。初期状態は何も選択されていませんから、空リストです。引数 k は順列の要素として選択すべき要素の個数です。`get_perm(l, s, k)` の再帰呼び出しの間不変です。`get_perm(l, s, k)` の本体のプログラミングについて説明します。

```
    if len(s)==k:
        r.append(s)
        return
```

の部分で、引数 s に k 個の要素がセットされているかチェックしています。 k 個の要素がセットされていれば、順列が一つ求まったので、s を r に追加して `get_perm(l, s, k)` の呼び出し元に帰ります。 k 個の要素がセットされていなければ、更に l から次の要素を選択する必要があります。その処理が

```
    for e in l:
        old_s = s[:]
        s.append(e)
        new_l = l[:]
        new_l.remove(e)
        get_perm(new_l, s, k)
        s = old_s
```

の部分です。基本的にはリスト l から順番に要素 e を取り出し、s に e を追加し、l から e を取り除き、新しい引数 l と s で get_perm(l, s, k) を呼び出しています。残りの部分は get_perm(l, s, k) から戻った場合に、l と s を復元するための処理をしています。

```
 old_s = s[:]
s.append(e)
```

では、s の現在の値を復元するために old_s に完全にコピーし、s に e を追加しています。

```
new_l = l[:]
new_l.remove(e)
```

では、l から e を取り除くと復元するのが厄介ですから、new_l に l を完全コピーし、l 二は手を付けず、new_l から e を取り除き、new_l と s を get_perm(l, s, k) の引数として、再帰呼び出しをしています。get_perm(l, s, k) から戻ってくると、

```
s = old_s
```

で、s の復元をしています。再帰呼び出しは慣れないとなかなか使いこなせないですが、これが自由に使いこなせるようになるとプログラミングの幅が大幅に拡大します。プログラミングの本質は再帰呼び出しの理解であると言っても過言ではないと思います。

重複順列のプログラムは

```
l = []
r = []
def get_duplication(l, s, k):
    global r
    if len(s)==k:
        r.append(s)
        return
    for e in l:
        old_s = s[:]
        s.append(e)
        get_duplication(l, s, k)
        s = old_s
n = int(input('n='))
k = int(input('k='))
l = [i for i in range(1, n+1)]
get_duplication(l, [], k)
print(len(r))
print(r)
```

です。実行結果は

```
n=3
k=3
27
[[1, 1, 1], [1, 1, 2], [1, 1, 3], [1, 2, 1], [1, 2, 2], [1, 2, 3],
```

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

です。順列のプログラムとの違いは関数 get_duplication(l, s, k) です。

```
    for e in l:
        old_s = s[:]
        s.append(e)
        get_duplication(l, s, k)
        s = old_s
```

の部分で、重複していいので l から e を取り除く必要がありません。違いはそれだけです。

組み合わせのプログラムは

```
l = []
r = []
def get_combination(l, s, k):
    global r
    if len(s)==k:
        r.append(s)
        return
    new_l = l[:]
    for e in l:
        old_s = s[:]
        s.append(e)
        new_l.remove(e)
        get_combination(new_l, s, k)
        s = old_s
n = int(input('n='))
k = int(input('k='))
l = [i for i in range(1, n+1)]
get_combination(l, [], k)
print(len(r))
print(r)
```

で与えられる。実行結果は

```
n=5
k=3
10
[[1, 2, 3], [1, 2, 4], [1, 2, 5], [1, 3, 4], [1, 3, 5], [1, 4, 5], [2, 3, 4],
 [2, 3, 5], [2, 4, 5], [3, 4, 5]]
```

です。順列のプログラムとの違いは関数 get_combination(l, s, k) です。

```
    new_l = l[:]
    for e in l:
        old_s = s[:]
        s.append(e)
        new_l.remove(e)
        get_combination(new_l, s, k)
        s = old_s
```

の部分で、一旦選択された e はその後、組み合わせの要素として選択されませんから、代入文 new_l = l[:] が for 文の外に出ています。追加すべき要素のリストは段々小さくなっていきます。違いはそれだけです。

重複組み合わせのプログラムは

```
l = []
r = []
def get_duplication(l, s, k):
    global r
    if len(s)==k:
        r.append(s)
        return
    new_l = l[:]
    for e in l:
        old_s = s[:]
        s.append(e)
        get_duplication(new_l, s, k)
        new_l.remove(e)
        s = old_s
n = int(input('n='))
k = int(input('k='))
l = [i for i in range(1, n+1)]
get_duplication(l, [], k)
print(len(r))
print(r)
```

です。実行結果は

```
n=4
k=3
20
[[1, 1, 1], [1, 1, 2], [1, 1, 3], [1, 1, 4], [1, 2, 2], [1, 2, 3],
[1, 2, 4], [1, 3, 3], [1, 3, 4], [1, 4, 4], [2, 2, 2], [2, 2, 3],
[2, 2, 4], [2, 3, 3], [2, 3, 4], [2, 4, 4], [3, 3, 3], [3, 3, 4],
 [3, 4, 4], [4, 4, 4]]
```

です。組み合わせのプログラムとの違いは関数 get_duplication(l, s, k) です。

```
    new_l = l[:]
    for e in l:
        old_s = s[:]
        s.append(e)
        get_duplication(new_l, s, k)
        new_l.remove(e)
        s = old_s
```

の部分で、一旦選択された e は直後に再び使うことが可能なので、関数 `get_duplication(l, s, k)` の再帰呼び出しの前には e を削除せず、戻った後で削除しています。

順列を求めるプログラムを作ったのでそれを使って、行列式を定義通り計算してみましょう。必要な関数を順番に作っていきます。順列を計算する関数は作っているので、偶置換か奇置換かを判定する関数 sgn(J) を作ります。

順列 $J = \{j_1, j_2, j_3, \cdots, j_n\}$ において、$m = 1.2, \cdots, n-1$ に対して項 j_m の後ろにある項 $j_l, m+1 \leq l \leq n$ のうちで j_m より値が小さい項の個数 $p(j_m)$ の総和

$$p(J) = p(j_1) + p(j_2) + p(j_3) + \cdots + p(j_{n-1})$$

が偶数であれば偶置換で、奇数であれば奇置換なのでこれを使うことにします。

J が偶置換であれば 1, 奇置換であれば -1　を返す関数 sgn(J) は

```
def sgn(J):
    s = 0
    for m in range(len(J)-1):
        for l in range(m+1, len(J)):
            if J[m] > J[l]: s += 1
    if  s % 2 == 0: return 1 else: return -1
```

で定義できます。

n次正方行列 A の行列式は A の (i,j) 成分を a_{ij} とすると

$$det(A) = \sum_{J \in P_n} sign(J) a_{1j_1} a_{2j_2} a_{3j_3} \cdots a_{nj_n}$$

です。つまり。$[1, 2, 3, \cdots, n]$ のすべての順列の集合 P_n を作り、各順列 $J = [j_1, j_2, j_3, \cdots, j_n]$ に対して

$$sign([j_1, j_2, j_3, \cdots, j_n]) a_{1j_1} a_{2j_2} a_{3j_3} \cdots a_{nj_n}$$

を計算し、それらの和が行列式であるが我々が普通習う行列式の定義です。

行列式を計算するプログラムは

```
l = []
r = []
def get_perm(l, s, k):
    global r
    if len(s)==k:
        r.append(s)
        return
```

```
    for e in l:
        old_s = s[:]
        s.append(e)
        new_l = l[:]
        new_l.remove(e)
        get_perm(new_l, s, k)
        s = old_s
A = eval(input('行列 : '))
n = len(A)
k = len(A)
l = [i for i in range(0, n)]
get_perm(l, [], k)
def sgn(J):
    s = 0
    for m in range(len(J)-1):
        for l in range(m+1, len(J)):
            if J[m] > J[l]: s += 1
    if  s % 2 == 0: return 1
    else: return -1
d = 0
for J in r:
    s = 1
    k = 0
    for e in A:
        s *= e[J[k]]
        k += 1
    d += s * sgn(J)
print(d)
```

です。実行すると

```
行列 : [[1,2,3,4],[0,1,1,1],[1,0,2,1],[3,1,3,2]]
-6
```

です。ここで

```
l = [i for i in range(0, n)]
```

と、順列を生成する元のリストを $l = [0, 1, 2, \cdots, n-1]$ と修正して使っています。

行列式を行列の成分とその余因子の積の和にすることを余因子展開すると言います。余因子展開を使って、帰納的に行列式を計算してみましょう。

n 次正方行列 A から (i,j) 成分の余因子 A_{ij} を作るためのアルゴリズムは

1. 行列 A から i 行と j 列をそれぞれ抜き去った、一回り小さい行列を用意する
2. 用意した行列の行列式を計算する。（これを D_{ij} と書きます）

3. 上の行列式に、行番号と列番号の和が奇数ならば「マイナス」を、偶数ならば「プラス」をつけて完成！

です。

行列式はこの予因子を使って、

$$|A| = a_{1j}A_{1j} + a_{2j}A_{2j} + \cdots + a_{nj}A_{nj} \qquad (1 \leq j \leq n)$$

$$|A| = a_{i1}A_{i1} + a_{i2}A_{i2} + \cdots + a_{in}A_{in} \qquad (1 \leq i \leq n)$$

と書くことが出来ます。これを使って行列式を帰納的に計算してみましょう。

まず必要なのが、余因子を計算する関数です。行列式を計算する関数 det(A) が定義されていると仮定します。函数 cofactor(A, i, j) を定義する必要があります。まず、行列 A から i 行と j 列をそれぞれ抜き去った、一回り小さい行列を計算しなければなりません。i 行を削除するのは簡単で

```
del A[i]
```

でよいです。j 列を削除するのは、for 文を使って削除します。

```
def cofactor(A, i, j):
    del A[i]
    for e in A:
        del e[j]
A = eval(input("行列 : "))
i = 1
j = 2
cofactor(A, i, j)
print(A)
```

を実行してみます。

```
行列 : [[1,2,3],[4,5,6],[7,8,9]]
[[1, 2], [7, 8]]
```

です。基本的な部分ができたので、関数 det(A) を定義することが出来ます。

```
import copy
def cofactor(A, i, j):
    del A[i]
    for e in A:
        del e[j]
    a = det(A)
    if (i+j)%2 == 0:
        return a
    else:
        return -a
def det(A):
```

```
    if len(A) == 1: return A[0][0]
    d = 0
    for j in range(len(A)):
        new_A = copy.deepcopy(A)
        d += cofactor(new_A, 0, j) * A[0][j]
    return d
A = eval(input("行列 : "))
print(det(A))
```

を実行してみます。

```
行列 : [[1,2,3,4],[0,1,1,1],[1,0,2,1],[3,1,3,2]]
-6
```

です。プログラムを見ていきましょう。関数 det() の定義は

```
def det(A):
    if len(A) == 1: return A[0][0]
    d = 0
    for j in range(len(A)):
        new_A = copy.deepcopy(A)
        d += cofactor(new_A, 0, j) * A[0][j]
    return d
```

です。

```
    if len(A) == 1: return A[0][0]
```

で、1 次正方行列の時の行列式の値の決定は簡単です。これが再帰呼び出しの停止条件です。本当は 2 次正方行列の時の行列式の値の決定も簡単なので、再帰呼び出しせずに直接計算した方が速いと思います。

```
    d = 0
    for j in range(len(A)):
        new_A = copy.deepcopy(A)
        d += cofactor(new_A, 0, j) * A[0][j]
    return d
```

A が 1 次正方行列でない時は、この部分を実行します。公式

$$|A| = a_{11}A_{11} + a_{12}A_{12} + \cdots + a_{1n}A_{1n}$$

を使って、計算しています。関数 cofactor(A, i, j) で A が変更されるので、A にはコピーを渡す必要があります。コピーするのに

```
        new_A = A[:]
```

としたのでは、これは浅いコピーといわれるもので、リストのリストである A のようなリストではコピーが不十分です。このような場合は深いコピーと呼ばれる

```
        new_A = copy.deepcopy(A)
```

のような deepcopy() 関数による処理が必要です。この為に

```
import copy
```

のインポート文が必要です。関数 cofactor() の定義は

```
def cofactor(A, i, j):
    del A[i]
    for e in A:
        del e[j]
    a = det(A)
    if (i+j)%2 == 0:
        return a
    else:
        return -a
```

です。これは行列 A の i 行、j 列を削除した一回り小さくなった行列の行列を det() で計算し、符号を付して返しているだけです。思ったよりもコンパクトなプログラムにすることが出来ました。

行列式が計算できるようになたので、連立方程式をクラメール (Cramer) の公式を使って計算してみましょう。クラメール (Cramer) の公式というのは

$$\begin{cases} 2x + 4y = 10 & (1) \\ x + 3y = 6 & (2) \end{cases}$$

は、行列を使って

$$\begin{pmatrix} 2 & 4 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 10 \\ 6 \end{pmatrix}$$

と表現して、

$$x = \frac{\begin{vmatrix} 10 & 4 \\ 6 & 3 \end{vmatrix}}{\begin{vmatrix} 2 & 4 \\ 1 & 3 \end{vmatrix}} \quad y = \frac{\begin{vmatrix} 2 & 10 \\ 1 & 6 \end{vmatrix}}{\begin{vmatrix} 2 & 4 \\ 1 & 3 \end{vmatrix}}$$

を計算しなさいと言うものです。行列式を計算する関数は作っているので、行列の要素を入れ替える関数を作れば完成です。行列 A の j 列をベクトル b で置き換え、その行列式を計算する、関数 `det_change(A, b, j)` を作ります。

```
def det_change(A, b, j):
    k = 0
    for e in A:
        e[j] = b[k]
        k += 1
    return det(A)
```

でいいです。全体のプログラムは

```
import copy
def cofactor(A, i, j):
    del A[i]
    for e in A:
        del e[j]
    a = det(A)
    if (i+j)%2 == 0:
        return a
    else:
        return -a
def det(A):
    if len(A) == 1: return A[0][0]
    d = 0
    for j in range(len(A)):
        new_A = copy.deepcopy(A)
        d += cofactor(new_A, 0, j) * A[0][j]
    return d
def det_change(A, b, j):
    k = 0
    for e in A:
        e[j] = b[k]
        k += 1
    return det(A)



A = eval(input("行列 : "))
b = eval(input(' ベクトル : '))
a = det(A)
r = []
for j in range(len(A)):
    new_A = copy.deepcopy(A)
    new_b = b[:]
    r.append(det_change(new_A, new_b, j)/a)
print(r)
```

です。実行すると

```
行列 : [[2,4],[1,3]]
ベクトル : [10,6]
[3.0, 1.0]
```

です。

これは NumPy を使えば、次のようにすればよい。

```
import numpy as np
A = np.array([[1, 2, -1], [2, 4 , 1], [3, 8, 1]])
b = np.array([-3, 0, -3])
x = np.linalg.solve(A, b)
print( x )
```

実行すると

```
[ 1. -1.  2.]
```

となります。

Jason Rosenhouse and Laura Taalman 著「**Taking SUDOKU Seriously**」（翻訳もありますが、英語で読んでも難しくはないです）には、数独に関連した色々は数学の問題を取り上げ解説しています。その中に数独の解の個数を求める問題があります。これは厄介な問題で、数独の解の集合上の変換群による同値類だけでも **5,472,730,538** 個あることがコンピュータを使って証明されています。この本では数独を簡単にした、**4**×**4** の盤の **Shidoku** というパズルを使ってのその計算方法の考え方が載っています。ここでは、この本に載っていないその解説を理解するためのプログラムを作ってみましょう。**Sidoku** の解は各列、各行、２×２の **4** つの各ブロックに１，２，３，４が一個ずつ配置されたものです。

```
1 2 | 3 4     1 2 | 3 4     1 2 | 3 4
3 4 | 1 2     3 4 | 2 1     3 4 | 1 2
---------     ---------     ---------
2 1 | 4 3     2 1 | 4 3     2 3 | 4 1
4 3 | 2 1     4 3 | 1 2     4 1 | 2 3

1 2 | 4 3     1 2 | 4 3     1 2 | 4 3
3 4 | 1 2     3 4 | 2 1     3 4 | 2 1
---------     ---------     ---------
2 1 | 3 4     2 3 | 3 4     2 1 | 1 4
4 3 | 2 1     4 1 | 1 2     4 3 | 3 2

1 2 | 3 4     1 2 | 3 4     1 2 | 3 4
3 4 | 1 2     3 4 | 1 2     3 4 | 2 1
---------     ---------     ---------
4 1 | 2 3     4 3 | 2 1     4 3 | 1 2
2 3 | 4 1     2 1 | 4 3     2 1 | 4 3

1 2 | 4 3     1 2 | 4 3     1 2 | 4 3
3 4 | 2 1     3 4 | 1 2     3 4 | 2 1
---------     ---------     ---------
4 1 | 3 2     4 3 | 2 1     4 3 | 1 2
2 3 | 1 4     2 1 | 3 4     2 1 | 3 4
```

は、左上隅のブロックの数字が固定された **12** 個の解です。これがすべてであることは手計算で簡単に確かめることが出来ます。**Shidoku** の解の全体は、これらに **Relabeling the digits**（数

字の入れ替え）を施した $\mathbf{12 \times 4 ! = 288}$ 個です。

Sidoku の解であるかどうかは

```
def ShidokuP(s):
    for k in range(4):
        for n in range(1, 5):
            if n not in s[k]:
                return False
    for i in range(4):
        for n in range(1, 5):
            flag2 = False
            for k in range(4):
                if n == s[k][i]:
                    flag2 = True
                    break
            if not flag2:
                return False
    for kk in range(2):
        for ii in range(2):
            for n in range(1, 5):
                flag2 = False
                for k in range(2*kk, 2*(kk+1)):
                    for i in range(2*ii, 2*(ii+1)):
                        if n == s[k][i]:
                            flag2 = True
                            break
                    if flag2:
                        break
                if not flag2:
                    return False
    return True

s = [[1,2,3,4],[3,4,1,2],[2,1,4,3],[4,3,2,1]]
print(ShidokuP(s))
s1 = [[1,2,3,4],[3,4,1,2],[2,1,4,3],[4,1,2,1]]
print(ShidokuP(s1))
s2 = [[1,2,3,4],[3,4,1,2],[2,1,4,3],[1,3,2,4]]
print(ShidokuP(s2))
s3 = [[1,3,2,4],[3,4,1,2],[2,1,4,3],[4,2,3,4]]
print(ShidokuP(s3))
```

のプログラムでチェックできます。

Shidoku の解の全数は **288** 個で、

1. **Relabeling the digits**

2. **Permuting the rows in a band or the columns in a pillar**

3. **Permuting the blocks in a given band or pillar**

4. **Any rotation or reflection**

の変換で同値類を取ると **Shidoku** の解の同値類は

```
1 2 | 3 4     1 2 | 3 4
3 4 | 1 2     3 4 | 1 2
---------     ---------
2 1 | 4 3     2 3 | 4 1
4 3 | 2 1     4 1 | 2 3
```

の二つの同値類だけで、その同値類に含まれる解の個数はそれぞれ、**96** 個と **192** 個であると書いています。この事実を証明するには、Shidoku の解の全数が 288 個であることは手計算で理解できますが、後半の同値類の話は、膨大な計算になり、「Taking SUDOKU Seriously」では

1. Relabeling the digits

2. Permuting the rows in a band or the columns in a pillar

3. Permuting the blocks in a given band or pillar

4. Any rotation or reflection

の変換で生成される変換群が Shidoku の解の集合に作用し、その orbits （軌道）の個数が 2 である事を Burnside's lemma を適用して計算すればよいことを解説しています。更に、ダイレクトに Burnside's lemma を適用しなくても、Relabeling the digits （数字の入れ替え）を除いた変換で生成される変換群の元の個数は 128 個であると説明し、左上隅のブロックの数字が固定された 12 個の解の集合にこの部分群を作用させ、Relabeling the digits （数字の入れ替え）の効果を考慮した固定点の数を数え、 Burnside's lemma を適用して

$$\frac{56(0)+48(2)+9(4)+4(6)+6(8)+4(10)+1(12)}{128}=\frac{256}{128}=2$$

の計算で、orbits （軌道）の個数が 2 である事を概説しています。これらのことを Python のプログラムを作ることにより追体験してみましょう。

左上隅のブロックの数字が固定された 12 個の解

```
1 2 | 3 4     1 2 | 3 4     1 2 | 3 4
3 4 | 1 2     3 4 | 2 1     3 4 | 1 2
---------     ---------     ---------
2 1 | 4 3     2 1 | 4 3     2 3 | 4 1
4 3 | 2 1     4 3 | 1 2     4 1 | 2 3

1 2 | 4 3     1 2 | 4 3     1 2 | 4 3
3 4 | 1 2     3 4 | 2 1     3 4 | 2 1
---------     ---------     ---------
2 1 | 3 4     2 3 | 3 4     2 1 | 1 4
```

```
4 3 | 2 1     4 1 | 1 2     4 3 | 3 2

1 2 | 3 4     1 2 | 3 4     1 2 | 3 4
3 4 | 1 2     3 4 | 1 2     3 4 | 2 1
---------     ---------     ---------
4 1 | 2 3     4 3 | 2 1     4 3 | 1 2
2 3 | 4 1     2 1 | 4 3     2 1 | 4 3

1 2 | 4 3     1 2 | 4 3     1 2 | 4 3
3 4 | 2 1     3 4 | 1 2     3 4 | 2 1
---------     ---------     ---------
4 1 | 3 2     4 3 | 2 1     4 3 | 1 2
2 3 | 1 4     2 1 | 3 4     2 1 | 3 4
```

は、手計算でどきますが、コンピュータで計算するには、ShidokuP(s) を使って、次のようなにプログラミングすれば良いです。

```
import copy

def ShidokuP(s):
    for k in range(4):
        for n in range(1, 5):
            if n not in s[k]:
                return False
    for i in range(4):
        for n in range(1, 5):
            flag2 = False
            for k in range(4):
                if n == s[k][i]:
                    flag2 = True
                    break
            if not flag2:
                return False
    for kk in range(2):
        for ii in range(2):
            for n in range(1, 5):
                flag2 = False
                for k in range(2*kk, 2*(kk+1)):
                    for i in range(2*ii, 2*(ii+1)):
                        if n == s[k][i]:
                            flag2 = True
                            break
                    if flag2:
                        break
```

```
                if not flag2:
                    return False
    return True

def get_basic_squares(s, S):
    def is_fill(s):
        for k in range(4):
            for i in range(4):
                if s[k][i] == 0:
                    return (k, i)
        return (4, 4)

    (k, i) = is_fill(s)
    if (k, i) == (4, 4):
        if ShidokuP(s):
            if s not in S:
                S.append(s)
            return
        return
    for n in range(1, 5):
        if n in s[k]: continue
        ss = copy.deepcopy(s)
        ss[k][i] = n
        get_basic_squares(ss, S)

S = []
s = [[1,2,0,0],[3,4,0,0],[0,0,0,0],[0,0,0,0]]
get_basic_squares(s, S)
print(len(S))
print(S)
```

実行すると

```
12
[[[1, 2, 3, 4], [3, 4, 1, 2], [2, 1, 4, 3], [4, 3, 2, 1]],
[[1, 2, 3, 4], [3, 4, 1, 2], [2, 3, 4, 1], [4, 1, 2, 3]],
[[1, 2, 3, 4], [3, 4, 1, 2], [4, 1, 2, 3], [2, 3, 4, 1]],
[[1, 2, 3, 4], [3, 4, 1, 2], [4, 3, 2, 1], [2, 1, 4, 3]],
[[1, 2, 3, 4], [3, 4, 2, 1], [2, 1, 4, 3], [4, 3, 1, 2]],
[[1, 2, 3, 4], [3, 4, 2, 1], [4, 3, 1, 2], [2, 1, 4, 3]],
[[1, 2, 4, 3], [3, 4, 1, 2], [2, 1, 3, 4], [4, 3, 2, 1]],
[[1, 2, 4, 3], [3, 4, 1, 2], [4, 3, 2, 1], [2, 1, 3, 4]],
[[1, 2, 4, 3], [3, 4, 2, 1], [2, 1, 3, 4], [4, 3, 1, 2]],
[[1, 2, 4, 3], [3, 4, 2, 1], [2, 3, 1, 4], [4, 1, 3, 2]],
```

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

と表示します。Shidoku のすべての解を求めるには

```
import copy

def ShidokuP(s):
    for k in range(4):
        for n in range(1, 5):
            if n not in s[k]:
                return False
    for i in range(4):
        for n in range(1, 5):
            flag2 = False
            for k in range(4):
                if n == s[k][i]:
                    flag2 = True
                    break
            if not flag2:
                return False
    for kk in range(2):
        for ii in range(2):
            for n in range(1, 5):
                flag2 = False
                for k in range(2*kk, 2*(kk+1)):
                    for i in range(2*ii, 2*(ii+1)):
                        if n == s[k][i]:
                            flag2 = True
                            break
                    if flag2:
                        break
                if not flag2:
                    return False
    return True

def get_basic_squares(s, S):
    def is_fill(s):
        for k in range(4):
            for i in range(4):
                if s[k][i] == 0:
                    return (k, i)
        return (4, 4)
```

```
    (k, i) = is_fill(s)
    if (k, i) == (4, 4):
        if ShidokuP(s):
            if s not in S:
                S.append(s)
            return
        return
    for n in range(1, 5):
        if n in s[k]: continue
        ss = copy.deepcopy(s)
        ss[k][i] = n
        get_basic_squares(ss, S)

S = []
s = [[1,2,0,0],[3,4,0,0],[0,0,0,0],[0,0,0,0]]
get_basic_squares(s, S)
##print(len(S))
##print(S)

def perm(n, k):
    l = [i for i in range(1, n+1)]
    r = []
    def get_perm(l, s, k):
        if len(s)==k:
            r.append(s)
            return
        for e in l:
            old_s = s[:]
            s.append(e)
            new_l = l[:]
            new_l.remove(e)
            get_perm(new_l, s, k)
            s = old_s

    get_perm(l, [], k)
    return r

P = perm(4,4)
##print(len(P))
##print(P)

def get_all_squares(P, S):
    SS = []
```

```
    def change_cells(p, s):
        ss =[[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
        for k in range(4):
            for i in range(4):
                ss[k][i] = p[s[k][i]-1]
        return ss
    for s in S:
        for p in P:
            ss = change_cells(p, s)
            if ss not in SS:
                SS.append(ss)
    return SS

SS = get_all_squares(P, S)
print(len(SS))
print(SS)
```

実行する良いです。わざわざ、コンピュータで確かめることはないですが、確かに、288 個あることが分かります。必要ならそれらすべての配置がリスト SS にセットされています。

次に、変換群を確定することは厄介なので、後に回し、何も考えずに力任せで、Shidoku の解の同値類は

```
1 2 | 3 4     1 2 | 3 4
3 4 | 1 2     3 4 | 1 2
---------     ---------
2 1 | 4 3     2 3 | 4 1
4 3 | 2 1     4 1 | 2 3
```

の二つの同値類だけで、その同値類に含まれる解の個数はそれぞれ、96 個と 192 個であることをチェックすることが出来るプログラムを考えてみます。

「1 Relabeling the digits」（数字の入れ替え） でどれだけ同値な解ができるかチェックするには、次のようなプログラミングをすれば良いです。

```
def perm(n, k):
    l = [i for i in range(1, n+1)]
    r = []
    def get_perm(l, s, k):
        if len(s)==k:
            r.append(s)
            return
        for e in l:
            old_s = s[:]
            s.append(e)
            new_l = l[:]
            new_l.remove(e)
```

```
            get_perm(new_l, s, k)
            s = old_s

    get_perm(l, [], k)
    return r

P = perm(4,4)

s1 = [[1,2,3,4],[3,4,1,2],[2,1,4,3],[4,3,2,1]]
s2 = [[1,2,3,4],[3,4,1,2],[2,3,4,1],[4,1,2,3]]

def get_perm_equiv(s, P, S):
    def change_cells(p, s):
        ss =[[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
        for k in range(4):
            for i in range(4):
                ss[k][i] = p[s[k][i]-1]
        return ss
    for p in P:
        ss = change_cells(p, s)
        if ss not in S:
            S.append(ss)

##def change_cells(p, s):
##    ss =[[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
##    for k in range(4):
##        for i in range(4):
##            ss[k][i] = p[s[k][i]-1]
##    return ss
##p1 = [4, 3, 2, 1]
##ss = change_cells(p1, s1)
##print(s1)
##print(ss)

S1 = []
get_perm_equiv(s1, P, S1)
print(len(S1))
get_perm_equiv(s2, P, S2)
print(len(S2))
```

で、共に 24 と表示します。

さらに、「**2 Permuting the rows in a band or the columns in a pillar**」（帯内の行の入れ替えと柱内の列の入れ替え） を追加することで、どれだけ同値な解ができるかチェックするは、

次のようなプログラミングすれば良いです。

```
def perm(n, k):
    l = [i for i in range(1, n+1)]
    r = []
    def get_perm(l, s, k):
        if len(s)==k:
            r.append(s)
            return
        for e in l:
            old_s = s[:]
            s.append(e)
            new_l = l[:]
            new_l.remove(e)
            get_perm(new_l, s, k)
            s = old_s

    get_perm(l, [], k)
    return r

P = perm(4,4)
##print(len(P))
##print(P)
s1 = [[1,2,3,4],[3,4,1,2],[2,1,4,3],[4,3,2,1]]
s2 = [[1,2,3,4],[3,4,1,2],[2,3,4,1],[4,1,2,3]]

def get_perm_equiv(s, P, S):
    def change_cells(p, s):
        ss =[[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
        for k in range(4):
            for i in range(4):
                ss[k][i] = p[s[k][i]-1]
        return ss
    for p in P:
        ss = change_cells(p, s)
        if ss not in S:
            S.append(ss)

##def change_cells(p, s):
##    ss =[[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
##    for k in range(4):
##        for i in range(4):
##            ss[k][i] = p[s[k][i]-1]
```

```
##    return ss
##p1 = [4, 3, 2, 1]
##ss = change_cells(p1, s1)
##print(s1)
##print(ss)

##S1 = []
##get_perm_equiv(s1, P, S1)
##print(len(S1))
##S2 = []
##get_perm_equiv(s2, P, S2)
##print(len(S2))

def band(n, s):
    ss = copy.deepcopy(s)
    if n == 0:
        return ss
    elif n == 1:
        ss[0] = s[1]
        ss[1] = s[0]
        return ss
    elif n == 2:
        ss[2] = s[3]
        ss[3] = s[2]
        return ss
    else:
        ss[0] = s[1]
        ss[1] = s[0]
        ss[2] = s[3]
        ss[3] = s[2]
        return ss
##ss = band(3, s1)
##print(s1)
##print(ss)

def pillar(n, s):
    ss = copy.deepcopy(s)
    if n == 0:
        return ss
    elif n == 1:
        for k in range(4):
            ss[k][0] = s[k][1]
            ss[k][1] = s[k][0]
```

```
        return ss
    elif n == 2:
        for k in range(4):
            ss[k][2] = s[k][3]
            ss[k][3] = s[k][2]
        return ss
    else:
        for k in range(4):
            ss[k][0] = s[k][1]
            ss[k][1] = s[k][0]
            ss[k][2] = s[k][3]
            ss[k][3] = s[k][2]
        return ss
##ss = pillar(3, s1)
##print(s1)
##print(ss)

def band_pillar(S):
    SS = copy.deepcopy(S)
    for s in S:
        for n in range(4):
            for m in range(4):
                ss = band(n, s)
                sss = pillar(m, ss)
                if sss not in SS:
                    SS.append(sss)
    return SS

S1 = []
get_perm_equiv(s1, P, S1)
SS1 = band_pillar(S1)
print(len(SS1))
S2 = []
get_perm_equiv(s1, P, S2)
SS2 = band_pillar(S2)
print(len(SS2))
```

で、共に 96 と表示します。

さらに、「**3 Permuting the blocks in a given band or pillar**」（帯の入れ替えと柱の入れ替え） を追加することで、どれだけ同値な解ができるかチェックするは、次のようなプログラミングすれば良いです。

```
def perm(n, k):
    l = [i for i in range(1, n+1)]
```

```
    r = []
    def get_perm(l, s, k):
        if len(s)==k:
            r.append(s)
            return
        for e in l:
            old_s = s[:]
            s.append(e)
            new_l = l[:]
            new_l.remove(e)
            get_perm(new_l, s, k)
            s = old_s

    get_perm(l, [], k)
    return r

P = perm(4,4)
##print(len(P))
##print(P)
s1 = [[1,2,3,4],[3,4,1,2],[2,1,4,3],[4,3,2,1]]
s2 = [[1,2,3,4],[3,4,1,2],[2,3,4,1],[4,1,2,3]]

def get_perm_equiv(s, P, S):
    def change_cells(p, s):
        ss =[[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
        for k in range(4):
            for i in range(4):
                ss[k][i] = p[s[k][i]-1]
        return ss
    for p in P:
        ss = change_cells(p, s)
        if ss not in S:
            S.append(ss)

##def change_cells(p, s):
##    ss =[[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
##    for k in range(4):
##        for i in range(4):
##            ss[k][i] = p[s[k][i]-1]
##    return ss
##p1 = [4, 3, 2, 1]
##ss = change_cells(p1, s1)
##print(s1)
```

```
##print(ss)

##S1 = []
##get_perm_equiv(s1, P, S1)
##print(len(S1))
##S2 = []
##get_perm_equiv(s2, P, S2)
##print(len(S2))

def band(n, s):
    ss = copy.deepcopy(s)
    if n == 0:
        return ss
    elif n == 1:
        ss[0] = s[1]
        ss[1] = s[0]
        return ss
    elif n == 2:
        ss[2] = s[3]
        ss[3] = s[2]
        return ss
    else:
        ss[0] = s[1]
        ss[1] = s[0]
        ss[2] = s[3]
        ss[3] = s[2]
        return ss
##ss = band(3, s1)
##print(s1)
##print(ss)

def pillar(n, s):
    ss = copy.deepcopy(s)
    if n == 0:
        return ss
    elif n == 1:
        for k in range(4):
            ss[k][0] = s[k][1]
            ss[k][1] = s[k][0]
        return ss
    elif n == 2:
        for k in range(4):
            ss[k][2] = s[k][3]
```

```
            ss[k][3] = s[k][2]
        return ss
    else:
        for k in range(4):
            ss[k][0] = s[k][1]
            ss[k][1] = s[k][0]
            ss[k][2] = s[k][3]
            ss[k][3] = s[k][2]
        return ss
##ss = pillar(3, s1)
##print(s1)
##print(ss)

def band_pillar(S):
    SS = copy.deepcopy(S)
    for s in S:
        for n in range(4):
            for m in range(4):
                ss = band(n, s)
                sss = pillar(m, ss)
                if sss not in SS:
                    SS.append(sss)
    return SS
##S1 = []
##get_perm_equiv(s1, P, S1)
##SS1 = band_pillar(S1)
##print(len(SS1))
##S2 = []
##get_perm_equiv(s1, P, S2)
##SS2 = band_pillar(S2)
##print(len(SS2))

def block_band(n, s):
    ss = copy.deepcopy(s)
    if n == 0:
        return ss
    elif n == 1:
        ss[0] = s[2]
        ss[1] = s[3]
        ss[2] = s[0]
        ss[3] = s[1]
        return ss
```

```
def block_pillar(n, s):
    ss = copy.deepcopy(s)
    if n == 0:
        return ss
    elif n == 1:
        for k in range(4):
            ss[k][0] = s[k][2]
            ss[k][1] = s[k][3]
            ss[k][2] = s[k][0]
            ss[k][3] = s[k][1]
        return ss

def block_band_pillar(S):
    SS = copy.deepcopy(S)
    for s in S:
        for n in range(2):
            for m in range(2):
                ss = block_band(n, s)
                sss = block_pillar(m, ss)
                if sss not in SS:
                    SS.append(sss)
    return SS

S1 = []
get_perm_equiv(s1, P, S1)
SS1 = band_pillar(S1)
SSS1 = block_band_pillar(SS1)
print(len(SSS1))
S2 = []
get_perm_equiv(s1, P, S2)
SS2 = band_pillar(S2)
SSS2 = block_band_pillar(SS2)
print(len(SSS2))
```

で、共に 96 と表示します。

最後に、「**4 Any rotation or reflection**」（回転と鏡映） を追加することで、どれだけ同値な解ができるかチェックするは、次のようなプログラミングすれば良いです。

```
def perm(n, k):
    l = [i for i in range(1, n+1)]
    r = []
    def get_perm(l, s, k):
        if len(s)==k:
            r.append(s)
```

```
            return
        for e in l:
            old_s = s[:]
            s.append(e)
            new_l = l[:]
            new_l.remove(e)
            get_perm(new_l, s, k)
            s = old_s

    get_perm(l, [], k)
    return r

P = perm(4,4)
##print(len(P))
##print(P)
s1 = [[1,2,3,4],[3,4,1,2],[2,1,4,3],[4,3,2,1]]
s2 = [[1,2,3,4],[3,4,1,2],[2,3,4,1],[4,1,2,3]]

def get_perm_equiv(s, P, S):
    def change_cells(p, s):
        ss =[[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
        for k in range(4):
            for i in range(4):
                ss[k][i] = p[s[k][i]-1]
        return ss
    for p in P:
        ss = change_cells(p, s)
        if ss not in S:
            S.append(ss)

##def change_cells(p, s):
##    ss =[[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
##    for k in range(4):
##        for i in range(4):
##            ss[k][i] = p[s[k][i]-1]
##    return ss
##p1 = [4, 3, 2, 1]
##ss = change_cells(p1, s1)
##print(s1)
##print(ss)

##S1 = []
##get_perm_equiv(s1, P, S1)
```

```
##print(len(S1))
##S2 = []
##get_perm_equiv(s2, P, S2)
##print(len(S2))

def band(n, s):
    ss = copy.deepcopy(s)
    if n == 0:
        return ss
    elif n == 1:
        ss[0] = s[1]
        ss[1] = s[0]
        return ss
    elif n == 2:
        ss[2] = s[3]
        ss[3] = s[2]
        return ss
    else:
        ss[0] = s[1]
        ss[1] = s[0]
        ss[2] = s[3]
        ss[3] = s[2]
        return ss
##ss = band(3, s1)
##print(s1)
##print(ss)

def pillar(n, s):
    ss = copy.deepcopy(s)
    if n == 0:
        return ss
    elif n == 1:
        for k in range(4):
            ss[k][0] = s[k][1]
            ss[k][1] = s[k][0]
        return ss
    elif n == 2:
        for k in range(4):
            ss[k][2] = s[k][3]
            ss[k][3] = s[k][2]
        return ss
    else:
        for k in range(4):
```

```
            ss[k][0] = s[k][1]
            ss[k][1] = s[k][0]
            ss[k][2] = s[k][3]
            ss[k][3] = s[k][2]
        return ss
##ss = pillar(3, s1)
##print(s1)
##print(ss)

def band_pillar(S):
    SS = copy.deepcopy(S)
    for s in S:
        for n in range(4):
            for m in range(4):
                ss = band(n, s)
                sss = pillar(m, ss)
                if sss not in SS:
                    SS.append(sss)
    return SS
##S1 = []
##get_perm_equiv(s1, P, S1)
##SS1 = band_pillar(S1)
##print(len(SS1))
##S2 = []
##get_perm_equiv(s1, P, S2)
##SS2 = band_pillar(S2)
##print(len(SS2))

def block_band(n, s):
    ss = copy.deepcopy(s)
    if n == 0:
        return ss
    elif n == 1:
        ss[0] = s[2]
        ss[1] = s[3]
        ss[2] = s[0]
        ss[3] = s[1]
        return ss

def block_pillar(n, s):
    ss = copy.deepcopy(s)
    if n == 0:
        return ss
```

```
    elif n == 1:
        for k in range(4):
            ss[k][0] = s[k][2]
            ss[k][1] = s[k][3]
            ss[k][2] = s[k][0]
            ss[k][3] = s[k][1]
        return ss

def block_band_pillar(S):
    SS = copy.deepcopy(S)
    for s in S:
        for n in range(2):
            for m in range(2):
                ss = block_band(n, s)
                sss = block_pillar(m, ss)
                if sss not in SS:
                    SS.append(sss)
    return SS

##S1 = []
##get_perm_equiv(s1, P, S1)
##SS1 = band_pillar(S1)
##SSS1 = block_band_pillar(SS1)
##print(len(SSS1))
##S2 = []
##get_perm_equiv(s1, P, S2)
##SS2 = band_pillar(S2)
##SSS2 = block_band_pillar(SS2)
##print(len(SSS2))

def rotation(n, s):
    def rot(s):
        ss = copy.deepcopy(s)
        for k in range(4):
            for i in range(4):
                ss[k][i] = s[i][3-k]
        return ss
    ss = copy.deepcopy(s)
    if n == 0:
        return ss
    elif n == 1:
        ss = rot(s)
        return ss
```

```
    elif n == 2:
        ss = rot(s)
        sss = rot(ss)
        return sss
    else:
        ss = rot(s)
        sss = rot(ss)
        ssss = rot(sss)
        return ssss

##ss = rotation(1, s1)
##print(s1)
##print(ss)
##ss = rotation(2, s1)
##print(ss)
##ss = rotation(3, s1)
##print(ss)

def reflection(n, s):
    def rot(s):
        ss = copy.deepcopy(s)
        for k in range(4):
            for i in range(4):
                ss[k][i] = s[i][3-k]
        return ss
    def ref(s):
        ss = copy.deepcopy(s)
        for k in range(4):
            for i in range(4):
                ss[k][i] = s[k][3-i]
        return ss

    if n == 0:
        ss = ref(s)
        return ss
    elif n == 1:
        ss = ref(s)
        sss = rot(ss)
        return sss
    elif n == 2:
        ss = ref(s)
        sss = rot(ss)
        ssss = rot(sss)
```

```
        return ssss
    else:
        ss = ref(s)
        sss = rot(ss)
        ssss = rot(sss)
        sssss = rot(ssss)
        return sssss

##ss = reflection(0, s1)
##print(s1)
##print(ss)
##ss = reflection(1, s1)
##print(ss)
##ss = reflection(2, s1)
##print(ss)
##ss = reflection(3, s1)
##print(ss)

def rotation_reflection(S):
    SS = copy.deepcopy(S)
    for s in S:
        for n in range(4):
            ss = rotation(n, s)
            if ss not in SS:
                SS.append(ss)
        for n in range(4):
            ss = reflection(n, s)
            if ss not in SS:
                SS.append(ss)
    return SS

S1 = []
get_perm_equiv(s1, P, S1)
SS1 = band_pillar(S1)
SSS1 = block_band_pillar(SS1)
SSSS1 = rotation_reflection(SSS1)
print(len(SSSS1))

S2 = []
get_perm_equiv(s2, P, S2)
SS2 = band_pillar(S2)
SSS2 = block_band_pillar(SS2)
SSSS2 = rotation_reflection(SSS2)
```

```
print(len(SSSS2))
```

で、それぞれ 96 と 192 と表示します。同値類に含まれる解の数が異なることに気付きます。合計が **288** で、**Shidoku** の解の総数と一致するので、同値類は **2** 個であることが分かります。同値類の代表元は

s1 = [[**1,2,3,4**],[**3,4,1,2**],[**2,1,4,3**],[**4,3,2,1**]] と
s2 = [[**1,2,3,4**],[**3,4,1,2**],[**2,3,4,1**],[**4,1,2,3**]] 、
すなわち、

```
1 2 | 3 4     1 2 | 3 4
3 4 | 1 2     3 4 | 1 2
---------     ---------
2 1 | 4 3     2 3 | 4 1
4 3 | 2 1     4 1 | 2 3
```

です。

更に、最後に

```
s3 = [[3,1,2,4],[4,2,1,3],[1,3,4,2],[2,4,3,1]]
s4 = [[1,4,2,3],[2,3,4,1],[3,2,1,4],[4,1,3,2]]
s5 = [[2,3,4,1],[1,4,2,3],[3,2,1,4],[4,1,3,2]]


print(s3 in SSSS1)
print(s3 in SSSS2)
print(s4 in SSSS1)
print(s4 in SSSS2)
print(s5 in SSSS1)
print(s5 in SSSS2)
```

を追加すると

```
True
False
False
True
False
True
```

と、s3, s4, s5 の同値類がどちらか調べることが出来ます。

ついでに、**Burnside's Lemma** による **orbits** の個数もコンピュータで計算してみましょう。

```
def G_band_pillar(s, S):
    SS = copy.deepcopy(S)
    for n in range(4):
        for m in range(4):
            ss = band(n, s)
            sss = pillar(m, ss)
```

```
            g = {k:sss[k // 4][k % 4] for k in range(16)}
            if g not in SS:
                SS.append(g)
    return SS

def G_block_band_pillar(s, S):
    SS = copy.deepcopy(S)
    for n in range(2):
        for m in range(2):
            ss = block_band(n, s)
            sss = block_pillar(m, ss)
            g = {k:sss[k // 4][k % 4] for k in range(16)}
            if g not in SS:
                SS.append(g)
    return SS

def G_rotation_reflection(s, S):
    SS = copy.deepcopy(S)
    for n in range(4):
        ss = rotation(n, s)
        g = {k:ss[k // 4][k % 4] for k in range(16)}
        if g not in SS:
            SS.append(g)
        for n in range(4):
            ss = reflection(n, s)
        g = {k:ss[k // 4][k % 4] for k in range(16)}
        if g not in SS:
            SS.append(g)
    return SS

G = []
s = [[0,1,2,3],[4,5,6,7],[8,9,10,11],[12,13,14,15]]
G1 = G_band_pillar(s, G)
G2 = G_block_band_pillar(s, G1)
SG = G_rotation_reflection(s, G2)
print(len(SG))
```

のプログラムで、relabeling（数字の入れ替え）以外の基本的な変換をリスト SG に求めています。

変換は Python のデータ構造である辞書を使って表現しています。即ち、

```
s = [[0,1,2,3],[4,5,6,7],[8,9,10,11],[12,13,14,15]]
```

を

```
ss = [[a0,a1,a2,a3],[a4,a5,a6,a7],[a8,a9,a10,a11],[a12,a13,a14,a15]]
```

に移す変換を

```
{0:a0, 1:a1, 2:a2, 3:a3, 4:a4, 5:a5, 6:a6, 7:a7, 8:a8, 9:a9, 10:a10, 11:a11, 12:a12, 13:a13, 14:a
```

という辞書で表現しています。続いて

```
def composition(g1, g2):
    g = {n:g2[g1[n]] for n in range(16)}
    return g

def G_composition(SG, G):
    for g1 in SG:
        for g2 in SG:
            for g3 in SG:
                g4 = composition(g1, g2)
                g = composition(g4, g3)
                if g not in G:
                    G.append(g)
    return G

Gr = []
Gr = G_composition(SG, Gr)
print(len(Gr))
```

で、SG の元で生成される部分群を求めています。元の個数は 128 個です。最後に

```
def group_act(s, g):
    ss = copy.deepcopy(s)
    for n in range(16):
        ss[g[n] // 4][g[n] % 4] = s[n // 4][n % 4]
    return ss

def change_cells(p, s):
    ss =[[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
    for k in range(4):
        for i in range(4):
            ss[k][i] = p[s[k][i]-1]
    return ss

id = [1, 2, 3, 4]

def burnsides_lemma(S, Gr):
    count = 0
    for g in Gr:
        for s in S:
            ss = group_act(s, g)
```

```
            if ss == s:
                count += 1
        for p in P:
            if p == id: continue
            for s in S:
                ss = group_act(s, g)
                sss = change_cells(p, ss)
                if sss == s:
                    count += 1
    return count

S = SSSS1 + SSSS2
n = burnsides_lemma(S, Gr)
print(n / len(Gr) / 24)
```

で、orbits（軌道）の個数を Burnside's Lemma で計算しています。relabeling による変換とそれ以外の幾何学的変換には共通点がなく、可換ですから、Shidoku の解の変換群は relabeling による変換のなす群とそれ以外の幾何学的変換のなす群の直積になります。確かに orbits の個数は 2 と表示されます。

ここまでに作ったプログラムの全体は

```
import copy

def perm(n, k):
    l = [i for i in range(1, n+1)]
    r = []
    def get_perm(l, s, k):
        if len(s)==k:
            r.append(s)
            return
        for e in l:
            old_s = s[:]
            s.append(e)
            new_l = l[:]
            new_l.remove(e)
            get_perm(new_l, s, k)
            s = old_s

    get_perm(l, [], k)
    return r

P = perm(4,4)
##print(len(P))
##print(P)
```

```
s1 = [[1,2,3,4],[3,4,1,2],[2,1,4,3],[4,3,2,1]]
s2 = [[1,2,3,4],[3,4,1,2],[2,3,4,1],[4,1,2,3]]

def get_perm_equiv(s, P, S):
    def change_cells(p, s):
        ss =[[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
        for k in range(4):
            for i in range(4):
                ss[k][i] = p[s[k][i]-1]
        return ss
    for p in P:
        ss = change_cells(p, s)
        if ss not in S:
            S.append(ss)

##def change_cells(p, s):
##    ss =[[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
##    for k in range(4):
##        for i in range(4):
##            ss[k][i] = p[s[k][i]-1]
##    return ss
##p1 = [4, 3, 2, 1]
##ss = change_cells(p1, s1)
##print(s1)
##print(ss)

##S1 = []
##get_perm_equiv(s1, P, S1)
##print(len(S1))
##S2 = []
##get_perm_equiv(s2, P, S2)
##print(len(S2))

def band(n, s):
    ss = copy.deepcopy(s)
    if n == 0:
        return ss
    elif n == 1:
        ss[0] = s[1]
        ss[1] = s[0]
        return ss
    elif n == 2:
        ss[2] = s[3]
```

```
        ss[3] = s[2]
        return ss
    else:
        ss[0] = s[1]
        ss[1] = s[0]
        ss[2] = s[3]
        ss[3] = s[2]
        return ss
##ss = band(3, s1)
##print(s1)
##print(ss)

def pillar(n, s):
    ss = copy.deepcopy(s)
    if n == 0:
        return ss
    elif n == 1:
        for k in range(4):
            ss[k][0] = s[k][1]
            ss[k][1] = s[k][0]
        return ss
    elif n == 2:
        for k in range(4):
            ss[k][2] = s[k][3]
            ss[k][3] = s[k][2]
        return ss
    else:
        for k in range(4):
            ss[k][0] = s[k][1]
            ss[k][1] = s[k][0]
            ss[k][2] = s[k][3]
            ss[k][3] = s[k][2]
        return ss
##ss = pillar(3, s1)
##print(s1)
##print(ss)

def band_pillar(S):
    SS = copy.deepcopy(S)
    for s in S:
        for n in range(4):
            for m in range(4):
                ss = band(n, s)
```

```
                sss = pillar(m, ss)
                if sss not in SS:
                    SS.append(sss)
    return SS
##S1 = []
##get_perm_equiv(s1, P, S1)
##SS1 = band_pillar(S1)
##print(len(SS1))
##S2 = []
##get_perm_equiv(s1, P, S2)
##SS2 = band_pillar(S2)
##print(len(SS2))

def block_band(n, s):
    ss = copy.deepcopy(s)
    if n == 0:
        return ss
    elif n == 1:
        ss[0] = s[2]
        ss[1] = s[3]
        ss[2] = s[0]
        ss[3] = s[1]
        return ss

def block_pillar(n, s):
    ss = copy.deepcopy(s)
    if n == 0:
        return ss
    elif n == 1:
        for k in range(4):
            ss[k][0] = s[k][2]
            ss[k][1] = s[k][3]
            ss[k][2] = s[k][0]
            ss[k][3] = s[k][1]
        return ss

def block_band_pillar(S):
    SS = copy.deepcopy(S)
    for s in S:
        for n in range(2):
            for m in range(2):
                ss = block_band(n, s)
                sss = block_pillar(m, ss)
```

```
                if sss not in SS:
                    SS.append(sss)
    return SS

##S1 = []
##get_perm_equiv(s1, P, S1)
##SS1 = band_pillar(S1)
##SSS1 = block_band_pillar(SS1)
##print(len(SSS1))
##S2 = []
##get_perm_equiv(s1, P, S2)
##SS2 = band_pillar(S2)
##SSS2 = block_band_pillar(SS2)
##print(len(SSS2))

def rotation(n, s):
    def rot(s):
        ss = copy.deepcopy(s)
        for k in range(4):
            for i in range(4):
                ss[k][i] = s[i][3-k]
        return ss
    ss = copy.deepcopy(s)
    if n == 0:
        return ss
    elif n == 1:
        ss = rot(s)
        return ss
    elif n == 2:
        ss = rot(s)
        sss = rot(ss)
        return sss
    else:
        ss = rot(s)
        sss = rot(ss)
        ssss = rot(sss)
        return ssss

##ss = rotation(1, s1)
##print(s1)
##print(ss)
##ss = rotation(2, s1)
##print(ss)
```

```
##ss = rotation(3, s1)
##print(ss)

def reflection(n, s):
    def rot(s):
        ss = copy.deepcopy(s)
        for k in range(4):
            for i in range(4):
                ss[k][i] = s[i][3-k]
        return ss
    def ref(s):
        ss = copy.deepcopy(s)
        for k in range(4):
            for i in range(4):
                ss[k][i] = s[k][3-i]
        return ss

    if n == 0:
        ss = ref(s)
        return ss
    elif n == 1:
        ss = ref(s)
        sss = rot(ss)
        return sss
    elif n == 2:
        ss = ref(s)
        sss = rot(ss)
        ssss = rot(sss)
        return ssss
    else:
        ss = ref(s)
        sss = rot(ss)
        ssss = rot(sss)
        sssss = rot(ssss)
        return sssss

##ss = reflection(0, s1)
##print(s1)
##print(ss)
##ss = reflection(1, s1)
##print(ss)
##ss = reflection(2, s1)
##print(ss)
```

```
##ss = reflection(3, s1)
##print(ss)

def rotation_reflection(S):
    SS = copy.deepcopy(S)
    for s in S:
        for n in range(4):
            ss = rotation(n, s)
            if ss not in SS:
                SS.append(ss)
        for n in range(4):
            ss = reflection(n, s)
            if ss not in SS:
                SS.append(ss)
    return SS

S1 = []
get_perm_equiv(s1, P, S1)
SS1 = band_pillar(S1)
SSS1 = block_band_pillar(SS1)
SSSS1 = rotation_reflection(SSS1)
##print(len(SSSS1))

S2 = []
get_perm_equiv(s2, P, S2)
SS2 = band_pillar(S2)
SSS2 = block_band_pillar(SS2)
SSSS2 = rotation_reflection(SSS2)
##print(len(SSSS2))

##s3 = [[3,1,2,4],[4,2,1,3],[1,3,4,2],[2,4,3,1]]
##s4 = [[1,4,2,3],[2,3,4,1],[3,2,1,4],[4,1,3,2]]
##s5 = [[2,3,4,1],[1,4,2,3],[3,2,1,4],[4,1,3,2]]
##
##print(s3 in SSSS1)
##print(s3 in SSSS2)
##print(s4 in SSSS1)
##print(s4 in SSSS2)
##print(s5 in SSSS1)
##print(s5 in SSSS2)

def G_band_pillar(s, S):
    SS = copy.deepcopy(S)
```

```
    for n in range(4):
        for m in range(4):
            ss = band(n, s)
            sss = pillar(m, ss)
            g = {k:sss[k // 4][k % 4] for k in range(16)}
            if g not in SS:
                SS.append(g)
    return SS

def G_block_band_pillar(s, S):
    SS = copy.deepcopy(S)
    for n in range(2):
        for m in range(2):
            ss = block_band(n, s)
            sss = block_pillar(m, ss)
            g = {k:sss[k // 4][k % 4] for k in range(16)}
            if g not in SS:
                SS.append(g)
    return SS

def G_rotation_reflection(s, S):
    SS = copy.deepcopy(S)
    for n in range(4):
        ss = rotation(n, s)
        g = {k:ss[k // 4][k % 4] for k in range(16)}
        if g not in SS:
            SS.append(g)
        for n in range(4):
            ss = reflection(n, s)
        g = {k:ss[k // 4][k % 4] for k in range(16)}
        if g not in SS:
            SS.append(g)
    return SS

G = []
s = [[0,1,2,3],[4,5,6,7],[8,9,10,11],[12,13,14,15]]
G1 = G_band_pillar(s, G)
G2 = G_block_band_pillar(s, G1)
SG = G_rotation_reflection(s, G2)
print(len(SG))

def composition(g1, g2):
    g = {n:g2[g1[n]] for n in range(16)}
```

```
    return g

def G_composition(SG, G):
    for g1 in SG:
        for g2 in SG:
            for g3 in SG:
                g4 = composition(g1, g2)
                g = composition(g4, g3)
                if g not in G:
                    G.append(g)
    return G

Gr = []
Gr = G_composition(SG, Gr)
print(len(Gr))

def group_act(s, g):
    ss = copy.deepcopy(s)
    for n in range(16):
        ss[g[n] // 4][g[n] % 4] = s[n // 4][n % 4]
    return ss

def change_cells(p, s):
    ss =[[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
    for k in range(4):
        for i in range(4):
            ss[k][i] = p[s[k][i]-1]
    return ss

id = [1, 2, 3, 4]

def burnsides_lemma(S, Gr):
    count = 0
    for g in Gr:
        for s in S:
            ss = group_act(s, g)
            if ss == s:
                count += 1
        for p in P:
            if p == id: continue
            for s in S:
                ss = group_act(s, g)
                sss = change_cells(p, ss)
```

```
                if sss == s:
                    count += 1
    return count

S = SSSS1 + SSSS2
n = burnsides_lemma(S, Gr)
print(n / len(Gr) / 24)
```

です。

最後に、Relabeling the digits（数字の入れ替え）を除いた変換で生成される変換群の元の個数は128個であると説明し、左上隅のブロックの数字が固定された12個の解の集合にこの部分群を作用させ、Relabeling the digits（数字の入れ替え）の効果を考慮した固定点の数を数え、Burnside's lemma を適用して、

$$\frac{56(0)+48(2)+9(4)+4(6)+6(8)+4(10)+1(12)}{128}=\frac{256}{128}=2$$

の計算で、orbits（軌道）の個数が2である事を概説している、これをコンピュータでチェックしてみましょう。

上で作った関数を組み合わせれば良いです。

```
import copy

def ShidokuP(s):
    for k in range(4):
        for n in range(1, 5):
            if n not in s[k]:
                return False
    for i in range(4):
        for n in range(1, 5):
            flag2 = False
            for k in range(4):
                if n == s[k][i]:
                    flag2 = True
                    break
            if not flag2:
                return False
    for kk in range(2):
        for ii in range(2):
            for n in range(1, 5):
                flag2 = False
                for k in range(2*kk, 2*(kk+1)):
                    for i in range(2*ii, 2*(ii+1)):
                        if n == s[k][i]:
                            flag2 = True
                            break
```

```
                    if flag2:
                        break
                if not flag2:
                    return False
    return True

def get_basic_squares(s, S):
    def is_fill(s):
        for k in range(4):
            for i in range(4):
                if s[k][i] == 0:
                    return (k, i)
        return (4, 4)

    (k, i) = is_fill(s)
    if (k, i) == (4, 4):
        if ShidokuP(s):
            if s not in S:
                S.append(s)
            return
        return
    for n in range(1, 5):
        if n in s[k]: continue
        ss = copy.deepcopy(s)
        ss[k][i] = n
        get_basic_squares(ss, S)


def perm(n, k):
    l = [i for i in range(1, n+1)]
    r = []
    def get_perm(l, s, k):
        if len(s)==k:
            r.append(s)
            return
        for e in l:
            old_s = s[:]
            s.append(e)
            new_l = l[:]
            new_l.remove(e)
            get_perm(new_l, s, k)
            s = old_s
```

```
    get_perm(l, [], k)
    return r

P = perm(4,4)

def band(n, s):
    ss = copy.deepcopy(s)
    if n == 0:
        return ss
    elif n == 1:
        ss[0] = s[1]
        ss[1] = s[0]
        return ss
    elif n == 2:
        ss[2] = s[3]
        ss[3] = s[2]
        return ss
    else:
        ss[0] = s[1]
        ss[1] = s[0]
        ss[2] = s[3]
        ss[3] = s[2]
        return ss

def pillar(n, s):
    ss = copy.deepcopy(s)
    if n == 0:
        return ss
    elif n == 1:
        for k in range(4):
            ss[k][0] = s[k][1]
            ss[k][1] = s[k][0]
        return ss
    elif n == 2:
        for k in range(4):
            ss[k][2] = s[k][3]
            ss[k][3] = s[k][2]
        return ss
    else:
        for k in range(4):
            ss[k][0] = s[k][1]
            ss[k][1] = s[k][0]
            ss[k][2] = s[k][3]
```

```
            ss[k][3] = s[k][2]
        return ss

def band_pillar(S):
    SS = copy.deepcopy(S)
    for s in S:
        for n in range(4):
            for m in range(4):
                ss = band(n, s)
                sss = pillar(m, ss)
                if sss not in SS:
                    SS.append(sss)
    return SS

def block_band(n, s):
    ss = copy.deepcopy(s)
    if n == 0:
        return ss
    elif n == 1:
        ss[0] = s[2]
        ss[1] = s[3]
        ss[2] = s[0]
        ss[3] = s[1]
        return ss

def block_pillar(n, s):
    ss = copy.deepcopy(s)
    if n == 0:
        return ss
    elif n == 1:
        for k in range(4):
            ss[k][0] = s[k][2]
            ss[k][1] = s[k][3]
            ss[k][2] = s[k][0]
            ss[k][3] = s[k][1]
        return ss

def rotation(n, s):
    def rot(s):
        ss = copy.deepcopy(s)
        for k in range(4):
            for i in range(4):
                ss[k][i] = s[i][3-k]
```

```
            return ss
        ss = copy.deepcopy(s)
        if n == 0:
            return ss
        elif n == 1:
            ss = rot(s)
            return ss
        elif n == 2:
            ss = rot(s)
            sss = rot(ss)
            return sss
        else:
            ss = rot(s)
            sss = rot(ss)
            ssss = rot(sss)
            return ssss

    def reflection(n, s):
        def rot(s):
            ss = copy.deepcopy(s)
            for k in range(4):
                for i in range(4):
                    ss[k][i] = s[i][3-k]
            return ss
        def ref(s):
            ss = copy.deepcopy(s)
            for k in range(4):
                for i in range(4):
                    ss[k][i] = s[k][3-i]
            return ss

        if n == 0:
            ss = ref(s)
            return ss
        elif n == 1:
            ss = ref(s)
            sss = rot(ss)
            return sss
        elif n == 2:
            ss = ref(s)
            sss = rot(ss)
            ssss = rot(sss)
            return ssss
```

```
    else:
        ss = ref(s)
        sss = rot(ss)
        ssss = rot(sss)
        sssss = rot(ssss)
        return sssss

def G_band_pillar(s, S):
    SS = copy.deepcopy(S)
    for n in range(4):
        for m in range(4):
            ss = band(n, s)
            sss = pillar(m, ss)
            g = {k:sss[k // 4][k % 4] for k in range(16)}
            if g not in SS:
                SS.append(g)
    return SS

def G_block_band_pillar(s, S):
    SS = copy.deepcopy(S)
    for n in range(2):
        for m in range(2):
            ss = block_band(n, s)
            sss = block_pillar(m, ss)
            g = {k:sss[k // 4][k % 4] for k in range(16)}
            if g not in SS:
                SS.append(g)
    return SS

def G_rotation_reflection(s, S):
    SS = copy.deepcopy(S)
    for n in range(4):
        ss = rotation(n, s)
        g = {k:ss[k // 4][k % 4] for k in range(16)}
        if g not in SS:
            SS.append(g)
        for n in range(4):
            ss = reflection(n, s)
        g = {k:ss[k // 4][k % 4] for k in range(16)}
        if g not in SS:
            SS.append(g)
    return SS
```

```
G = []
s = [[0,1,2,3],[4,5,6,7],[8,9,10,11],[12,13,14,15]]
G1 = G_band_pillar(s, G)
G2 = G_block_band_pillar(s, G1)
SG = G_rotation_reflection(s, G2)

def composition(g1, g2):
    g = {n:g2[g1[n]] for n in range(16)}
    return g

def G_composition(SG, G):
    for g1 in SG:
        for g2 in SG:
            for g3 in SG:
                g4 = composition(g1, g2)
                g = composition(g4, g3)
                if g not in G:
                    G.append(g)
    return G

Gr = []
Gr = G_composition(SG, Gr)

def group_act(s, g):
    ss = copy.deepcopy(s)
    for n in range(16):
        ss[g[n] // 4][g[n] % 4] = s[n // 4][n % 4]
    return ss

def change_cells(p, s):
    ss =[[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
    for k in range(4):
        for i in range(4):
            ss[k][i] = p[s[k][i]-1]
    return ss

id = [1, 2, 3, 4]

BS = []
s = [[1,2,0,0],[3,4,0,0],[0,0,0,0],[0,0,0,0]]
get_basic_squares(s, BS)
print(len(BS))
print(BS)
```

```
def burnsides_lemma2(S, Gr, P):
    count = 0
    for g in Gr:
        for s in S:
            gs = group_act(s, g)
            flag = False
            for p in P:
                ps = change_cells(p, s)
                if gs == ps:
                    count += 1
    return count
count = burnsides_lemma2(BS, Gr, P)
print(count/128)
```

実行すると確かに 2 と表示します。

$$\frac{56(0)+48(2)+9(4)+4(6)+6(8)+4(10)+1(12)}{128}=\frac{256}{128}=2$$

の計算で何をしているかを正確に知るためには、群論計算システム gap を使って計算します。gap は Windows10 用もありますが、使い物になりません。ubuntu で使うか、 Windows10 でどうしてもやりたければ、Oracle VM VirtualBox をインストールし、それに ubuntu をインストールして、更に、gap をインストールします。gap は 1 ギガバイトぐらいあるので、 ubuntu をインストールする時、デフォルトの 1 ギガバイトではなく、最低でも 2 ギガバイトのメモリーを取っておく必要があります。生成元が沢山あるので

```
shidoku := Group(
(1,5)(2,6)(3,7)(4,8),
(9,13)(10,14)(11,15)(12,16),
(1,2)(5,6)(9,10)(13,14),
(3,4)(7,8)(11,12)(15,16),
(1,9)(2,10)(3,11)(4,12)(5,13)(6,14)(7,15)(8,16),
(1,3)(5,7)(9,11)(13,15)(2,4)(6,8)(10,12)(14,16),
(1,4,16,13)(2,8,15,9)(3,12,14,5)(6,7,11,10),
(1,4)(2,3)(5,8)(6,7)(9,12)(10,11)(13,16)(14,15));
```

というファイルを shidoku.gap という名前で作っておきます。

gap を起動し、

```
gap> Read(''shidoku'.gap'');
gap>
```

で群 shidoku の定義を読み込み、

```
gap> Size(shidoku);
128
gap>
```

で、群 shidoku の位数（元の個数）が 128 であることが分かります。
　群の作用は共役類で同じ個数からなる軌道を作るので、群 shidoku の共役類の個数を求めてみます。

```
gap> NrConjugacyClasses(shidoku);
20
```

共役類は 20 個です。共役類の代表元は具体的には

```
gap> ConjugacyClasses(shidoku);
[()^G,
(9,13)(10,14)(11,15)(12,16)^G,
(3,4)(7,8)(9,13)(10,14)(11,16)(12,15)^G,
(2,5)(3,9)(4,13)(7,10)(8,14)(12,15)^G,
(2,5)(3,9,4,13)(7,10,8,14)(11,12,16,15)^G,
(1,2)(3,4)(5,6)(7,8)(9,10)(11,12)(13,14)(15,16)^G,
(1,2)(3,4)(5,6)(7,8)(9,14)(10,13)(11,16)(12,15)^G,
(1,2,6,5)(3,10,8,13)(4,14,7,9)(11,12,16,15)^G,
(1,3)(2,4)(5,7)(6,8)(9,11)(10,12)(13,15)(14,16)^G,
(1,3)(2,4)(5,7)(6,8)(9,15)(10,16)(11,13)(12,14)^G,
(1,3,2,4)(5,7,6,8)(9,11,10,12)(13,15,14,16)^G,
(1,3,2,4)(5,7,6,8)(9,15,10,16)(11,14,12,13)^G,
(1,3,11,9)(2,7,12,13)(4,15,10,5)(6,8,16,14)^G,
(1,3,11,10,6,8,16,13)(2,7,12,14,5,4,15,9)^G,
(1,6)(2,5)(3,8)(4,7)(9,14)(10,13)(11,16)(12,15)^G,
(1,7)(2,8)(3,5)(4,6)(9,15)(10,16)(11,13)(12,14)^G,
(1,7,2,8)(3,6,4,5)(9,15,10,16)(11,14,12,13)^G,
(1,11)(2,12)(3,9)(4,10)(5,15)(6,16)(7,13)(8,14)^G,
(1,11,5,15)(2,12,6,16)(3,9,7,13)(4,10,8,14)^G,
(1,11,6,16)(2,12,5,15)(3,10,8,13)(4,9,7,14)^G]
gap>
```

で求めることが出来ます。それぞれの共役類の元の個数は

```
gap> List(ConjugacyClasses(shidoku), x->Size(x));
[1,4,4,8,16,2,4,8,4,8,6,8,16,16,1,4,4,4,8,4]
gap>
```

です。
　群 shidoku の位数は 128 で、

```
1 2 | 3 4     1 2 | 3 4     1 2 | 3 4
3 4 | 1 2     3 4 | 2 1     3 4 | 1 2
---------     ---------     ---------
2 1 | 4 3     2 1 | 4 3     2 3 | 4 1
4 3 | 2 1     4 3 | 1 2     4 1 | 2 3
```

```
1 2 | 4 3     1 2 | 4 3     1 2 | 4 3
3 4 | 1 2     3 4 | 2 1     3 4 | 2 1
---------     ---------     ---------
2 1 | 3 4     2 3 | 3 4     2 1 | 1 4
4 3 | 2 1     4 1 | 1 2     4 3 | 3 2

1 2 | 3 4     1 2 | 3 4     1 2 | 3 4
3 4 | 1 2     3 4 | 1 2     3 4 | 2 1
---------     ---------     ---------
4 1 | 2 3     4 3 | 2 1     4 3 | 1 2
2 3 | 4 1     2 1 | 4 3     2 1 | 4 3

1 2 | 4 3     1 2 | 4 3     1 2 | 4 3
3 4 | 2 1     3 4 | 1 2     3 4 | 2 1
---------     ---------     ---------
4 1 | 3 2     4 3 | 2 1     4 3 | 1 2
2 3 | 1 4     2 1 | 3 4     2 1 | 3 4
```

の 12 個の解空間に対する不動点だけを計算するだけですから、上の Python プログラムでやったように、すべての群の元に対して、不動点の個数を求めても大したことではないですが、「数独」の変換群は位数が大きいし、基本的な解空間の個数も大きいので、共役類の代表元に対してのみ、計算する方が良いという判断です。共役類の代表元に対して計算した式が

$$\frac{56(0)+48(2)+9(4)+4(6)+6(8)+4(10)+1(12)}{128}=\frac{256}{128}=2$$

であった訳です。

本来の数独の同値類の個数の計算のためには、更に巧妙な技巧が必要みたいです。

「数式処理」や「統計処理」や「グラフや幾何学的図形の作図・作成」のソフトが使え、プログラミング言語でも色々なライブラリーが使えるような現在では、数学を理解するためのプログラミング学習と言っても、ライブラリーの単純な組み合わせ（単純な数値計算）ではできない、このようなプログラミングが出来ないと他人に認めてもらえないかとも思いますし、プログラミングを学ぶ人もこのようなプログラムでないと現代におけるプログラミングすることの意義が実感できなくて、面白くないと思います。しかし、ある程度、プログラミング言語の文法の基本を学び、アルゴリズムやデータ構造の基本を学び、それなりの訓練をしないとこのようなプログラミングを自分ではでっちあげることができないのではないか（最低でも **300** 時間は必要）と思われることが、数学の教員免許に必須の「コンピュータ」**2** 単位 **30** 時間（集中でやれば **3** 日間の講義）だけでそれなりの結果を出さなければいけない教える側としては悩みの種です。

π の計算にはマチンの公式

$$\frac{\pi}{4}=4\cdot\arctan\frac{1}{5}-\arctan\frac{1}{239}$$

を使います。これを使って円周率を計算してみます。

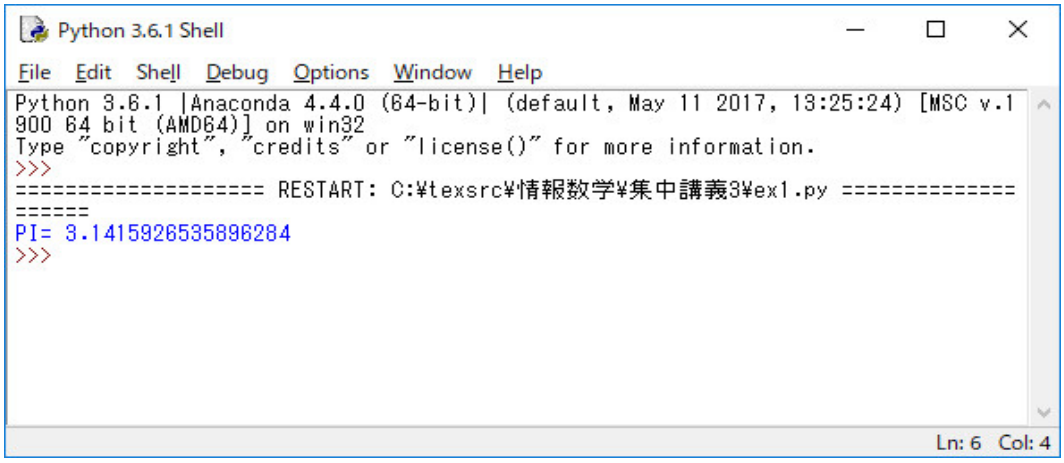
$\arctan x$ は

$$\arctan x=x-\frac{x^3}{3}+\frac{x^5}{5}-\frac{x^7}{7}+\frac{x^9}{9}-\frac{x^{11}}{11}+\cdots$$

で計算できます。

```
def arctan(x):
    s = 0.0
    n = 0
    while True:
        if n % 2 == 0:
            t = x ** (2*n+1) / (2*n+1)
        else:
            t = - x ** (2*n+1) / (2*n+1)
        if abs(t) < 0.1 ** 12:
            break;
        s += t
        n += 1
    return s
print( "PI=", 4*(4*arctan(1.0/5)-arctan(1.0/239)) )
```

を実行すると



です。

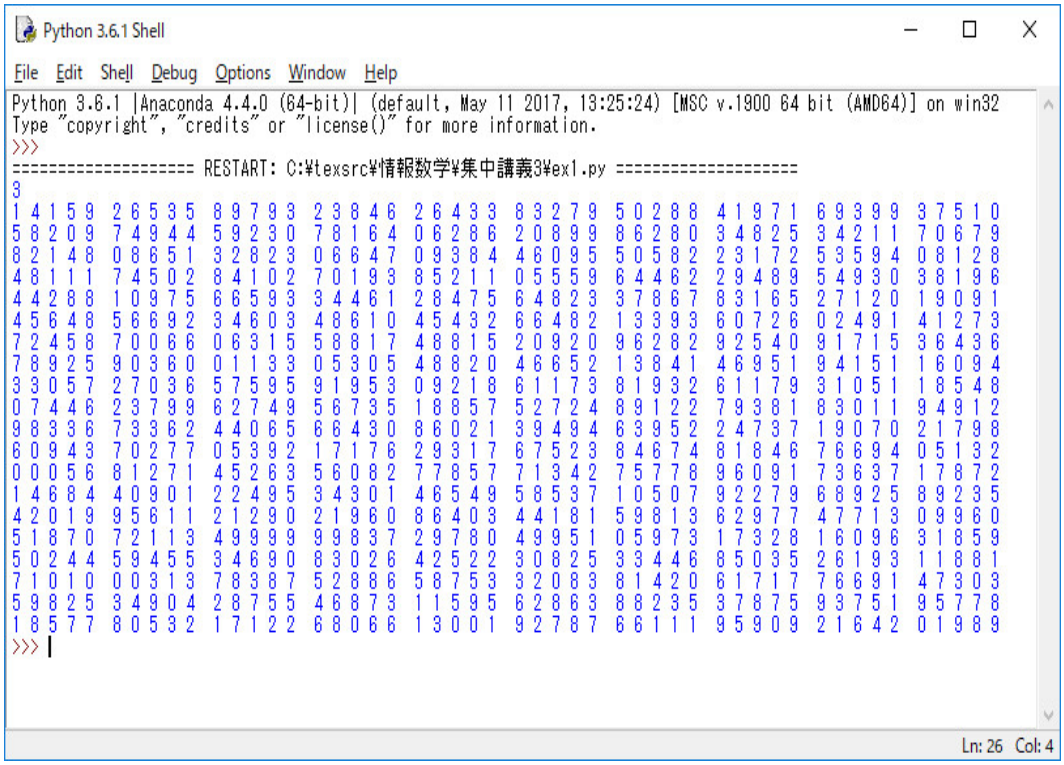
配列を使えば、大きな桁数の計算が出来るようになります。ここでは「**C** -言語とプログラミング-」米田信夫編　産業図書にある π の計算のプログラムを **Python** に翻訳します。

```
RADIX = 10
W = 1100

def init(a, n) :
    global W
    a.append(n)
    for i in range(1, W) :
        a.append(0)

def add(a, b, c) :
```

```
    global RADIX
    global W
    t = 0
    l = list(range(W))
    l.reverse()
    for i in l :
        x = b[i] + c[i] + t
        a[i] = x % RADIX
        t = x // RADIX
    if t != 0 :
        print( "overflow" )

def sub(a, b, c) :
    global RADIX
    global W
    t = 1
    l = list(range(W))
    l.reverse()
    for i in l :
        x = b[i] + (RADIX - 1 - c[i]) + t
        a[i] = x % RADIX
        t = x // RADIX
    if t != 1 :
        print( "overflow" )

def top(a, p) :
    global W
    while  p < W and a[p] == 0 :
        p += 1
    return p

def div(a, b, n) :
    global RADIX
    global W
    t = 0
    for i in range(W) :
        x = t * RADIX + b[i]
        a[i] = x // n
        t = x % n

def marctan(a, n, d) :
    global W
    e = []
```

```
    f = []

    init(a, 0)
    init(e, n)
    init(f, 0)
    div(e, e, d)
    p = top(e, 0)
    add(a, a, e)
    i = 3
    while  p < W :
        div(e, e, d)
        div(e, e, d)
        div(f, e, i)
        if  i % 4 == 1 :
            add(a, a, f)
        else:
            sub(a, a, f)
        p = top(e, p)
        i += 2

a = []
b = []
pi = []
marctan(a, 16, 5)
marctan(b, 4, 239)
init(pi, 0)
sub(pi, a, b)

print( "%d" % pi[0] )
for i in range(1, 1001) :
    print( "%d" % pi[i] , end=' ' )
    if  i % 5 == 0 : print( ' ' , end='' )
    if  i % 50 == 0 : print()
```

実行します。



π の小数点以下１０００桁は

3.
14159 26535 89793 23846 26433 83279 50288 41971 69399 37510
58209 74944 59230 78164 06286 20899 86280 34825 34211 70679
82148 08651 32823 06647 09384 46095 50582 23172 53594 08128
48111 74502 84102 70193 85211 05559 64462 29489 54930 38196
44288 10975 66593 34461 28475 64823 37867 83165 27120 19091
45648 56692 34603 48610 45432 66482 13393 60726 02491 41273
72458 70066 06315 58817 48815 20920 96282 92540 91715 36436
78925 90360 01133 05305 48820 46652 13841 46951 94151 16094
33057 27036 57595 91953 09218 61173 81932 61179 31051 18548
07446 23799 62749 56735 18857 52724 89122 79381 83011 94912
98336 73362 44065 66430 86021 39494 63952 24737 19070 21798
60943 70277 05392 17176 29317 67523 84674 81846 76694 05132
00056 81271 45263 56082 77857 71342 75778 96091 73637 17872
14684 40901 22495 34301 46549 58537 10507 92279 68925 89235
42019 95611 21290 21960 86403 44181 59813 62977 47713 09960
51870 72113 49999 99837 29780 49951 05973 17328 16096 31859
50244 59455 34690 83026 42522 30825 33446 85035 26193 11881
71010 00313 78387 52886 58753 32083 81420 61717 76691 47303
59825 34904 28755 46873 11595 62863 88235 37875 93751 95778
18577 80532 17122 68066 13001 92787 66111 95909 21642 01989

となります。

このπの計算にはマチンの公式

$$\frac{\pi}{4} = 4 \cdot \arctan\frac{1}{5} - \arctan\frac{1}{239}$$

を使います。$\arctan x$ は

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \frac{x^{11}}{11} + \cdots$$

で計算できます。従って

$$\pi = 16 \cdot \arctan\frac{1}{5} - 4 \cdot \arctan\frac{1}{239}$$

なので

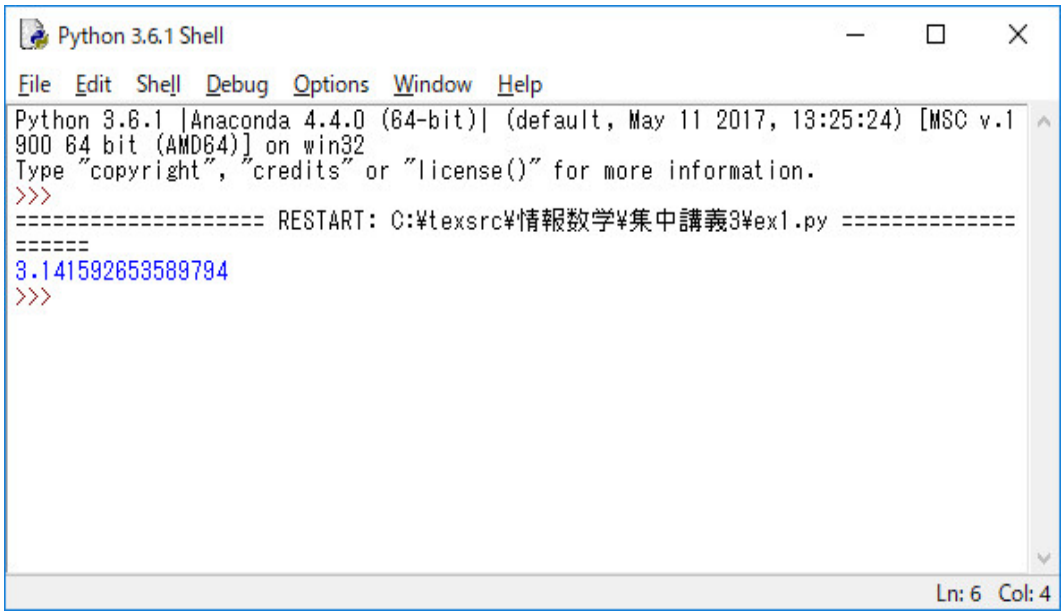
$$\begin{aligned}
\pi = \frac{16}{5} - \frac{\frac{16}{5}\cdot(\frac{1}{5})^2}{3} + \frac{\frac{16}{5}\cdot(\frac{1}{5})^2\cdot(\frac{1}{5})^2}{5} - \frac{\frac{16}{5}\cdot(\frac{1}{5})^2\cdot(\frac{1}{5})^2\cdot(\frac{1}{5})^2}{7} \\
+ \frac{\frac{16}{5}\cdot(\frac{1}{5})^2\cdot(\frac{1}{5})^2\cdot(\frac{1}{5})^2\cdot(\frac{1}{5})^2}{9} - \frac{\frac{16}{5}\cdot(\frac{1}{5})^2\cdot(\frac{1}{5})^2\cdot(\frac{1}{5})^2\cdot(\frac{1}{5})^2\cdot(\frac{1}{5})^2}{11} + \cdots \\
- \frac{4}{239} + \frac{\frac{4}{239}\cdot(\frac{1}{239})^2}{3} - \frac{\frac{4}{239}\cdot(\frac{1}{239})^2\cdot(\frac{1}{239})^2}{5} + \frac{\frac{4}{239}\cdot(\frac{1}{239})^2\cdot(\frac{1}{239})^2\cdot(\frac{1}{239})^2}{7} \\
- \frac{\frac{4}{239}\cdot(\frac{1}{239})^2\cdot(\frac{1}{239})^2\cdot(\frac{1}{239})^2\cdot(\frac{1}{239})^2}{9} + \frac{\frac{4}{239}\cdot(\frac{1}{239})^2\cdot(\frac{1}{239})^2\cdot(\frac{1}{239})^2\cdot(\frac{1}{239})^2\cdot(\frac{1}{239})^2}{11} - \cdots
\end{aligned}$$

を計算すれば良いことになります。

この式通り計算するプログラムは

```
def marctan(n, d) :
    a = 0.0
    e = float(n)
    f = 0.0
    e = e / d
    a = a + e
    i = 3
    while e > 0.1 ** 15:
        e = e / d
        e = e / d
        f = e / i
        if i % 4 == 1:
            a = a + f
        else:
            a = a - f
        i = i + 2
    return a
a = marctan(16, 5)
b = marctan(4, 239)
pi = a - b
print( "%.15f" % pi )
```

です。実行すると



です。

この計算を float ではなく、大きな桁数の固定小数点数で計算するためには、多きな桁数の数の足し算と引き算と割り算があれば良い事が分かります。更に、求める円周率の桁数がそれほど大きくなければ（$\arctan x$ を求めるための分子が int の範囲にあるならば)、割り算は多きな桁数の数を単独の整数で割る割り算で良い事が見て取れます。

足し算は

```
RADIX = 10
W = 1100

def add(a, b, c) :
    global RADIX
    global W
    t = 0
    l = list(range(W))
    l.reverse()
    for i in l :
        x = b[i] + c[i] + t
        a[i] = x % RADIX
        t = x // RADIX
    if t != 0 :
        print( "overflow" )
```

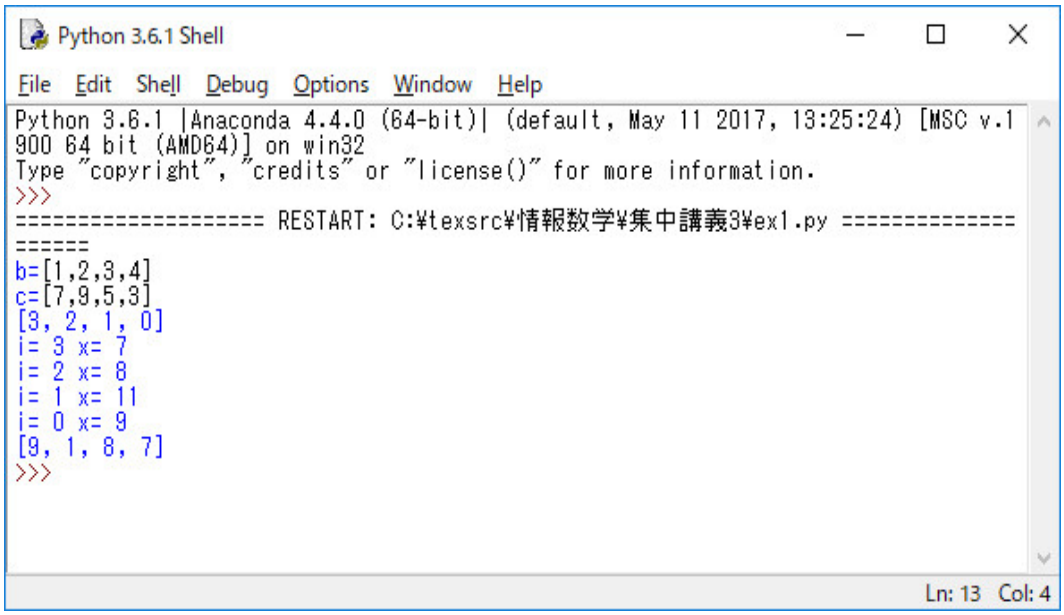
で、計算できます。

```
RADIX = 10
W = 1100
```

は、１０進数で、桁数は 1100 であることを示しています。これは１００進数でも 1000 進数でも良いですが、我々になじみ深い１０進数にしてあります。アルゴリズムは小学校で習う１０進数の足し算と同じものです。

```
def add(a, b, c) :
    global RADIX
    global W
    t = 0
    l = list(range(W))
    l.reverse()
    print( l )
    for i in l :
        x = b[i] + c[i] + t
        print( "i=", i, "x=", x )
        a[i] = x % RADIX
        t = x // RADIX
    if t != 0 :
        print( "overflow" )
RADIX = 10
W = 4
b = eval(input("b="))
c = eval(input("c="))
a = range(W)
add(a, b, c)
print( a )
```

を実行すると



です。これは、小数点の位置の想定を動かすことにより　$1234 + 7953 = 9187$ を計算したことにも、$123.4 + 795.3 = 918.7$ を計算したことにも、$1.234 + 7.953 = 9.187$ を計算したことにも使えます。

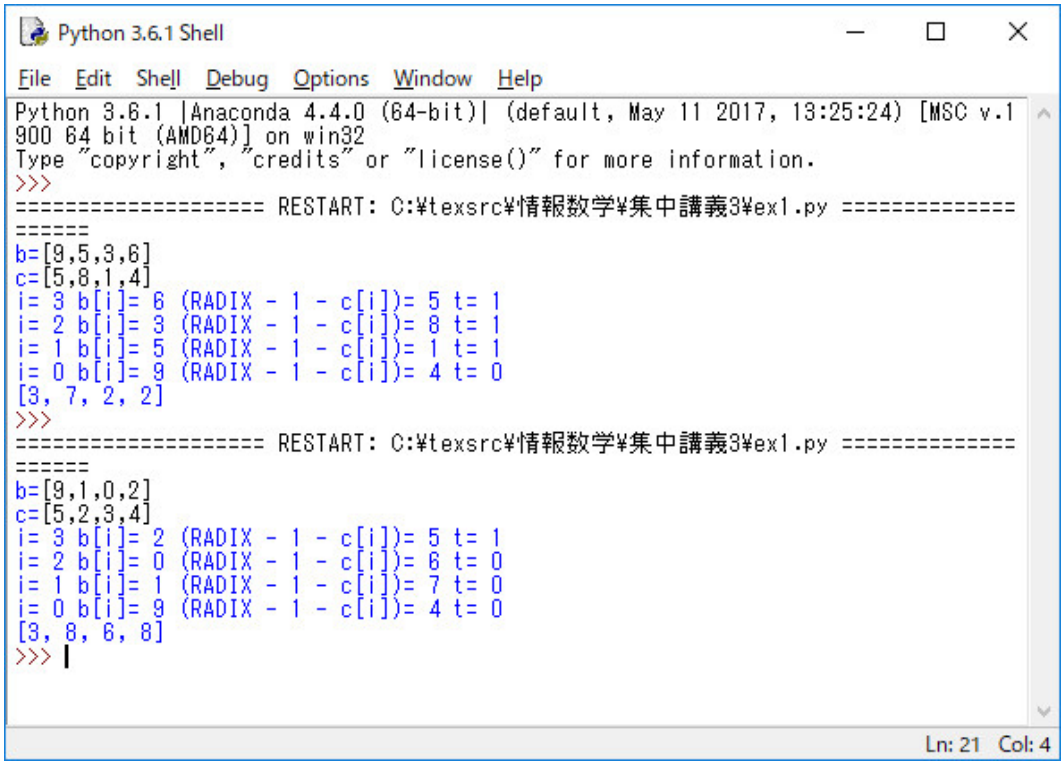
引き算は

```
RADIX = 10
W = 1100

def sub(a, b, c) :
    global RADIX
    global W
    t = 1
    l = list(range(W))
    l.reverse()
    for i in l :
        x = b[i] + (RADIX - 1 - c[i]) + t
        a[i] = x % RADIX
        t = x // RADIX
    if t != 1 :
        print( "overflow" )
```

で、計算できます。アルゴリズムは小学校で習う１０進数の引き算とは異なります。

```
RADIX = 10
W = 4
def sub(a, b, c) :
    global RADIX
    global W
    t = 1
    l = list(range(W))
    l.reverse()
    for i in l :
        x = b[i] + (RADIX - 1 - c[i]) + t
        print( "i=", i, "b[i]=", b[i], "(RADIX - 1 - c[i])=", (RADIX - 1 - c[i]),"t=", t)
        a[i] = x % RADIX
        t = x // RADIX
    if t != 1 :
        print( "overflow" )
b = eval(input("b="))
c = eval(input("c="))
a = list(range(W))
sub(a, b, c)
print( a )
```

を実行すると



です。確かに計算できています。小学校で習う１０進数のくり下がりの引き算は「減加法」とか「減々法」のどちらを教えるべきかの議論がありますが、上のような計算方法（アルゴリズム）もあります。機械語の講義で、負の数を表すのに２の補数というのを学びました。このアルゴリズムも同じことをしています。２進数の場合は２の補数 (1 と 0 を入れ替え 1 を足す。即ち、各数字を 1 から引き 1 を足す) でしたが、今は１０進数ですから１０の補数（各数字を 9 から引き 1 を足す）を使います。例えば、$9536-5814$ を計算するには、5814 を１０の補数 $4185+1$ にし、$9536+4186=13722$ を計算し、一番上の 1 を除いた 3722 が答えになります。$9102-3234$ の計算は 3234 の１０の補数 $6765+1$ を 9102 に加え $9102+6766=15868$ の一番上の 1 を除いた 5868 が答えになります。これが正しいのは、

$$9102-3234=9102-3234+10000-10000=9102+9999+1-3234-10000$$
$$=9102+(9999-3234)+1-10000=9102+(6765+1)-10000=15868-10000=5868$$

から、理解できます。このアルゴリズムでは、くり下がりがあるかどうかの判断（ if 文）が不要になります。

```
RADIX = 10
W = 1100 l

def init(a, n) :
    global W
    a.append(n)
    for i in range(1, W) :
        a.append(0)
```

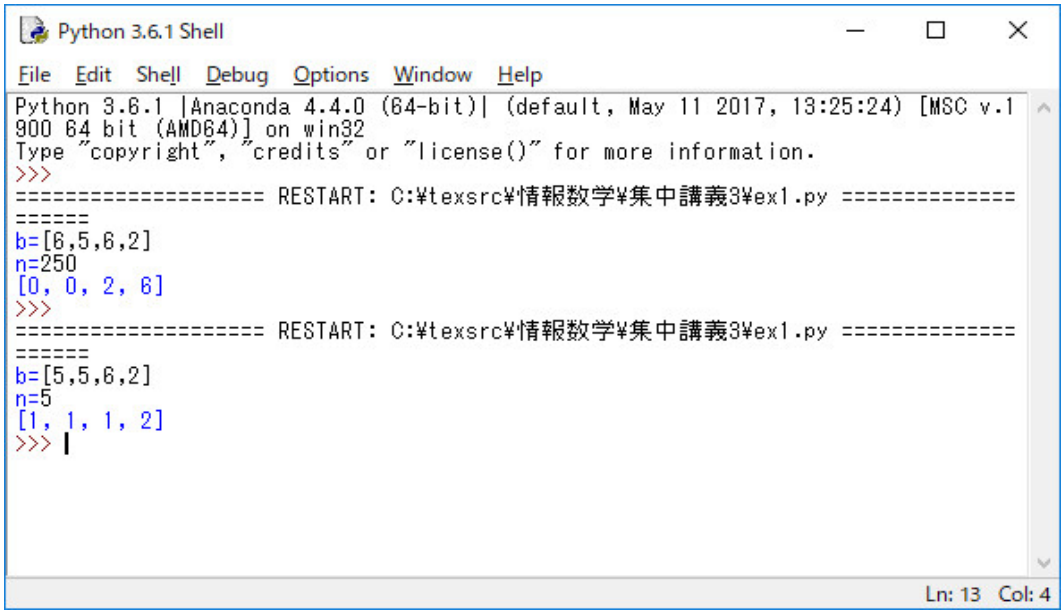
で、リスト a に、$a = [n, 0, \cdots, 0]$ をセットしています。
　割り算は

```
RADIX = 10
W = 1100

def div(a, b, n) :
    global RADIX
    global W
    t = 0
    for i in range(W) :
        x = t * RADIX + b[i]
        a[i] = x // n
        t = x % n
```

です。これは小学校で習う割り算と本質的に同じです。桁数の多い数を桁数の多い数で割る割り算は試行錯誤が必要で難しいです。割り算の効率の良いアルゴリズムは Knuth の「The Art of Computer Programming」 にアルゴリズムが載っています。

```
RADIX = 10
W = 4

def div(a, b, n) :
    global RADIX
    global W
    t = 0
    for i in range(W) :
        x = t * RADIX + b[i]
        a[i] = x // n
        t = x % n
b = eval(input("b="))
n = eval(input("n="))
a = list(range(W))
div(a, b, n)
print( a )
```

を実行すると



です。確かに計算できています。

```
RADIX = 10
W = 1100

def top(a, p) :
    global W
    while  p < W and a[p] == 0 :
        p += 1
    return p
```

で、リスト a の 0 でない最初のインデックスを求めています。これは、a が小さな数かを判定するのに使います。

これらの関数を使って

```
RADIX = 10
W = 1100

def marctan(a, n, d) :
    global W
    e = []
    f = []

    init(a, 0)
    init(e, n)
    init(f, 0)
    div(e, e, d)
    p = top(e, 0)
```

```
    add(a, a, e)
    i = 3
    while  p < W :
        div(e, e, d)
        div(e, e, d)
        div(f, e, i)
        if  i % 4 == 1 :
            add(a, a, f)
        else:
            sub(a, a, f)
        p = top(e, p)
        i += 2

a = []
b = []
pi = []
marctan(a, 16, 5)
marctan(b, 4, 239)
init(pi, 0)
sub(pi, a, b)

print( "%d" % pi[0] )
for i in range(1, 1001) :
    print( "%d" % pi[i] , end=' ' )
    if  i % 5 == 0 : print( ' ' , end='' )
    if  i % 50 == 0 : print()
```

で、円周率を計算しています。

ところで、Python には、クラス Decimal が定義されています。

```
from decimal import *
getcontext().prec = 1100

def marctan(n, d) :
    a = Decimal(0)
    e = Decimal(n)
    f = Decimal(0)
    e = e / d
    a = a + e
    i = 3
    while e > Decimal(0.1) ** 1100:
        e = e / d
        e = e / d
        f = e / i
```

```
        if i % 4 == 1:
            a = a + f
        else:
            a = a - f
        i = i + 2
    return a

if __name__ == '__main__':

    A = marctan(16, 5)
    B = marctan(4, 239)
    pi = A - B
    print(pi)
```

のプログラムで円周率が計算でき、一瞬で計算します。但し、高速で計算できるアルゴリズムが使われているので

```
>>> from decimal import *
>>> getcontext().prec = 30
>>> print(Decimal(120)-Decimal(120))
0
>>> print(Decimal(12)-Decimal(120))
-108
>>> print(Decimal(1.2)-Decimal(12.0))
-10.8000000000000000444089209850
>>> print(Decimal(1.1)*Decimal(1.1))
1.21000000000000019539925233403
>>> print(Decimal(12.0)-Decimal(0.120))
11.880000000000000044408920985
>>> print(Decimal(12.0)/Decimal(0.120))
100.000000000000003700743415417
>>>
```

のように、簡単な計算でも誤差が出ます。

π の計算の歴史 アルキメデス（紀元前 283-212 年）は $n = 6, 12, 24, 48, 96$ の正 n 角形の長さを計算し、半角の公式

$$\sin(\frac{x}{2}) = \pm\sqrt{\frac{1-\cos x}{2}}, \cos(\frac{x}{2}) = \pm\sqrt{\frac{1+\cos x}{2}}$$

を繰り返し使うことにより

$$3\frac{10}{71} < \pi < 3\frac{1}{7}$$

という評価を得ました。この値を改良しようとする中世のあらゆる試みは実を結びませんでしたが、ついに、アドリアン・ファン・ルーメンが（1580 年に）、アルキメデスの方法を使い、何年も

計算した末に小数点以下 20 桁までの値を得ることに成功しました。ルドルフ・ファン・ケーレンは（1596, 1616 年に）35 桁まで計算しました。この精度に達するために、ルドルフは $n = 62^{60}$ まで計算を続けなければなりませんでした。ドイツでは π のことをルドルフの数と言います。

日本では、江戸時代の１７１２年に関孝和が円周率を１１桁、１７２２年に建部賢弘が４１桁、さらに１７３９年刊の「方円算経」で松永良弼が５２桁計算しました。建部賢弘の円周率公式は

$$\frac{\pi^2}{9} = 1 + \frac{1^2}{3\cdot 4} + \frac{1^2\cdot 2^2}{3\cdot 4\cdot 5\cdot 6} + \frac{1^2\cdot 2^2\cdot 3^2}{3\cdot 4\cdot 5\cdot 6\cdot 7\cdot 8} + \cdots$$

で、松永良弼の円周率公式は

$$\frac{\pi}{3} = 1 + \frac{1^2}{4\cdot 6} + \frac{1^2\cdot 3^2}{4\cdot 6\cdot 8\cdot 10} + \frac{1^2\cdot 3^2\cdot 5^2}{4\cdot 6\cdot 8\cdot 10\cdot 12\cdot 14} + \cdots$$

です。

ライプニッツが公式

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \frac{x^{11}}{11} + \cdots$$

を発見します。この式で $x = 1$ と置けば、有名なライプニッツの公式

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - \cdots$$

（1682 年）が得られます。この式は美しさとは裏腹に、あまり役に立ちません。なにしろ、50 桁欲しければ 10^{50} 項の計算を「永遠に続ける努力で」（オイラー 1737)、行わねばなりません。ずっと有効なのは $\tan(\pi/6) = 1/\sqrt{3}$ を使うことです。すると

$$\pi = 2\sqrt{3}(1 - \frac{1}{3\cdot 3} + \frac{1}{5\cdot 3^2} - \frac{1}{7\cdot 3^3} + \cdots)$$

という公式が得られます。この式を用いて、Th.F. ド・ラニーは、「信じられないほどの仕事量を示している」210 項を加えることによって、127 桁（113 桁目に間違いがあった）を 1719 年に計算しました。

$$\tan(x+y) = \frac{\tan x + \tan y}{1 - \tan x \tan y}$$

の式に $u = \tan x, v = \tan y$ を代入すれば、$|\arctan u + \arctan v| < \pi/2$ のとき、

$$\arctan u + \arctan v = \arctan(\frac{u+v}{1-uv})$$

が得られます。ここで、$u = 1/2, v = 1/3$ を代入すれば、オイラーの公式 (1737 年)

$$\frac{\pi}{4} = \arctan\frac{1}{2} + \arctan\frac{1}{3}$$

が得られます。有名なジョン・マチンの公式は次のようにして求めます。$u = v = 1/5$ と置けば

$$2\cdot\arctan\frac{1}{5} = \arctan(\frac{2/5}{1-1/25}) = \arctan\frac{5}{12}$$

が得られます。$u = v = 5/12$ と置けば

$$2\cdot\arctan\frac{5}{12} = \arctan(\frac{5/6}{1-25/144}) = \arctan\frac{120}{119}$$

となります。そこで、$u = 120/119$ と置いて、v を

$$\frac{u+v}{1-uv} = 1 \text{ となるように、したがって } v = \frac{1-u}{1+u} = -\frac{1}{239}$$

とおきます。これらの式をまとめると、

$$\frac{\pi}{4} = 4 \cdot \arctan\frac{1}{5} - \arctan\frac{1}{239}$$

となります。この公式を使ってマチンは 100 桁の値を求めたことを W. ジョーンズ（1706 年）が細かい点は述べずに紹介しています。

最近のコンピュータによる円周率の計算では

$$\pi = 48 \cdot \arctan\frac{1}{49} - 128 \cdot \arctan\frac{1}{57} - 20 \cdot \arctan\frac{1}{239} + 48 \cdot \arctan\frac{1}{110443}$$

や

$$\pi = 176 \cdot \arctan\frac{1}{57} + 28 \cdot \arctan\frac{1}{239} - 48 \cdot \arctan\frac{1}{682} + 96 \cdot \arctan\frac{1}{12943}$$

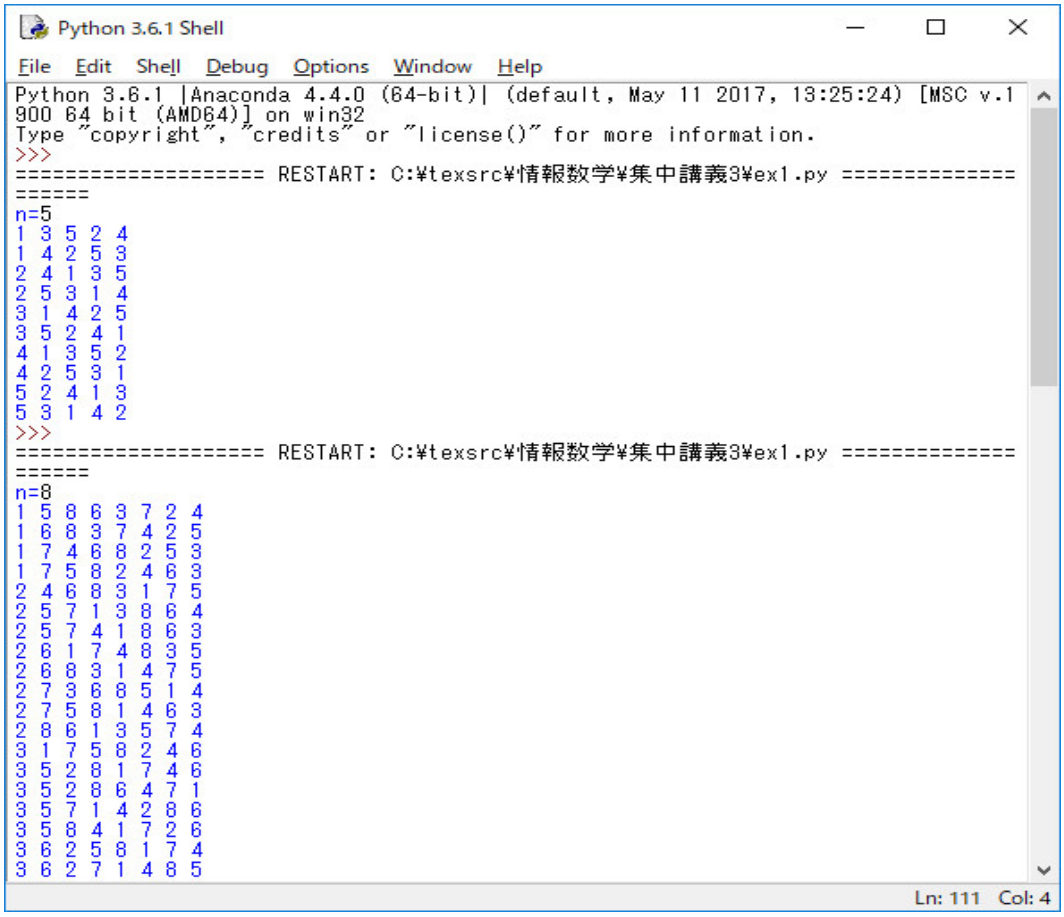
という式が使われたそうです。

最後に N 人の女王のプログラムを再帰を使った Python のプログラムを載せます。N は 8 以下です。

```
mm = []
ii = []
kk = []
ll = []

def queens(m, n) :
    global ii
    global kk
    global ll
    global mm

    for c in range(1, m+1) :
        if not (ii[c] == 0 and kk[n+1-c+m] == 0 and ll[n+1+c-1] == 0) :
            continue
        mm[n+1] = c
        ii[c] = kk[n+1-c+m] = ll[n+c] = 1
        if n+1 == m :
            for i in range(1, m+1) :
                print( "%d" % mm[i] , end=' ')
            print()
        else :
            queens(m, n+1)
        ii[c] = kk[n+1-c+m] = ll[n+c] = 0
n = int(input("n="))
for i in range(16) :
```

```
    kk.append(0)
    ll.append(0)
for i in range(9) :
    ii.append(0)
    mm.append(0)
queens(n, 0)
```

実行します。



ちなみに N 人の女王の **FORTRAN** のプログラムの例は次のようなものです。

```
      DIMENSION MM(8),JJ(8),KK(15),LL(15)
      DO 10 IX=1,8
   10    JJ(IX)=0
      DO 20 IX=1,15
         KK(IX)=0
   20    LL(IX)=0
      I=1
   30 N=0
   40 N=N+1
      IF(N.GT.8) GOTO 60
```

```
      IF(JJ(N).EQ.1) GOTO 40
      IF(KK(I-N+8).EQ.1) GOTO 40
      IF(LL(I+N-1).EQ.1) GOTO 40
      MM(I)=N
      JJ(N)=1
      KK(I-N+8)=1
      LL(I+N-1)=1
      I=I+1
      IF(I.LE.8) GOTO 30
      WRITE(6,50)(MM(IY),IY=1,8)
   50 FORMAT(5H ANS=,8I5)
   60 I=I-1
      IF(I.LT.1) STOP
      N=MM(I)
      JJ(N)=0
      KK(I-N+8)=0
      LL(I+N-1)=0
      GOTO 40
      END
```

FORTRAN では再帰が使えなかったので、理解するのが難しいプログラムになります。

演習問題

1. 三次元のベクトルをリストに入力し、その長さ（ノルム）を計算するプログラムを書け。
2. 三次元のベクトルを 2 個、リストに入力し、その内積を計算するプログラムを書け。
3. 三次正方行列をリストに入力し、その行列式を計算するプログラムを書け。
4. 二次正方行列を二個入力し、その積を計算するプログラムを書け。
5. 第 3 四分位数（中央値を除いた上位のデータの中央値）を計算するプログラムを書け。
6. 箱ひげ図を描くプログラムを書け。
7. 挿入ソートでは、空リストはすでに整列されていると考える。また、空でないリストは、残りのリストを整列した結果に。先頭の要素を挿入することで整列する。Haskell では

   ```
   insert x []  = [x]
   insert x (y : ys)  | x <= y     = x : y : ys
                      | otherwise = y : insert x ys

   isort []  = []
   isort (x : xs) = insert x (isort xs)
   ```

   とプログラミングできます。これを Python でプログラミングしなさい。

2.2 方程式の根の近似解法

昔の数学 B の教科書には正の数 a の平方根 $\sqrt{a}$ を求める次のようなアルゴリズムが載っている。

まず、縦の長さが 1、横の長さが a の長方形からはじめ面積 a を変えないで、ある規則にしたがって縦と横の長さだけ変え、正方形に近づけていく。正方形が得られたとき、その辺の長さを x とすれば、 $x^2 = a$ から平方根 $\sqrt{a}$ を求めることができる。長方形を正方形に近づける手順において、k 回目の縦、横の長さをそれぞれ y_k, x_k とする。縦と横の長さの平均

$$x_{k+1} = \frac{x_k + y_k}{2}$$

によって k+1 回目の横の長さを定める。$x_{k+1} \cdot y_{k+1} = a$ から縦の長さは

$$y_{k+1} = \frac{a}{x_{k+1}}$$

である。

$x_0 = a, y_0 = 1$ からはじめ、上の計算を続け、数列 $\{x_k\}, \{y_k\}$ を求め k を大きくしていくと、それぞれの数列が平方根 $\sqrt{a}$ に限りなく近づく。

この算法による BASIC のプログラムは、次のようになる。

```
PRINT "長方形から正方形による平方根の値"
INPUT "縦=1, 横の長さ A を入力 A=";A
PRINT "      X               Y"
X=A : Y = 1
FOR I=1 TO 5
  X=(X+Y)/2
  Y=A/X
  PRINT USING "0.0000000000000000" X, USING "0.0000000000000000" Y
NEXT I
END
```

実行すると

```
長方形から正方形による平方根の値
縦=1, 横の長さ A を入力 A=3
      X               Y
2.0000000000000000      1.5000000000000000
1.7500000000000000      1.7142857142857142
1.7321428571428572      1.7319587628865978
1.7320508100147274      1.7320508051230272
1.7320508075688772      1.7320508075688774
```

となる。わずか５回の繰り返しで、小数大１５位まで一致し、良い近似値が得られている。

なお、この算法では、平方根を求めるわけであるから、x_k, y_k の両方を求める必要はない。$y_k = \dfrac{a}{x_k}$ を $x_{k+1} = \dfrac{x_k + y_k}{2}$ に代入して

$$x_{k+1} = \frac{1}{2}(x_k + \frac{a}{x_k})$$

が得られる。これを使えばよい。これは数学 C にあるニュートン法の特別な場合である。

上の BASIC のプログラムを Python に翻訳すると次のようになる。

```
a = float(input('a= '))
x = a
y = 1.0
for i in range(6) :
    x = (x + y) / 2
    y = a / x
    print( x, "\t", y )
```

一般的に方程式の近似解を求めるには二分法とニュートン法がよく使われる。これは昔の数学 **C** で解説されている。一般的に方程式の近似解を求めるには二分法とニュートン法がよく使われる。これは数学 **C** で解説されている。次は二分法により平方根を求める旺文社数学 **C** の **BASIC** のプログラムである。

```
100 REM 二分法
110 EPS = 0.00001
120 INPUT "S=";S
130 DEF FNY(X)=X^2-S
140 IF S<1 THEN P=0:Q=1 ELSE P=1:Q=S
150 FOR N=1 TO 20
160   R=(P+Q)/2:NI=N
170   IF FNY(R)*FNY(Q)<0 THEN P=R ELSE Q=R
180   IF ABS(Q-P)<=EPS THEN 200
190 NEXT N
200 PRINT NI;"回反復　近似値＝";R
210 END
```

実行すると

```
S=3
18.0 回反復　近似値＝ 1.7320480346679688
```

となる。

s の平方根を求めるには $f(x) = x^2 - s = 0$ の解を求めればよい。s の値により $f(p) < 0,\ f(q) > 0$ なる p, q を初期値として与える。

Python で二分法により平方根を求めてみよう。s の平方根を求めるには $f(x) = x^2 - s = 0$ の解を求めればよい。s の値により $f(p) < 0,\ f(q) > 0$ なる p, q を初期値として与える。

```
import math

def fn(x, s) :
```

```
    return x * x - s

eps = 0.0000001
s = float(input("s= "))
if s < 1 :
    p = 0.0
    q = 1.0
else :
    p = 1.0
    q = s
for n in range(200) :
    r = (p + q) / 2
    if fn(r, s) * fn(q, s) < 0 :
        p = r
    else :
        q = r
    if math.fabs(q - p) < eps :
        break
print( n+1,  "回反復　近似値＝", p )
print( "p*p=", p*p )
```

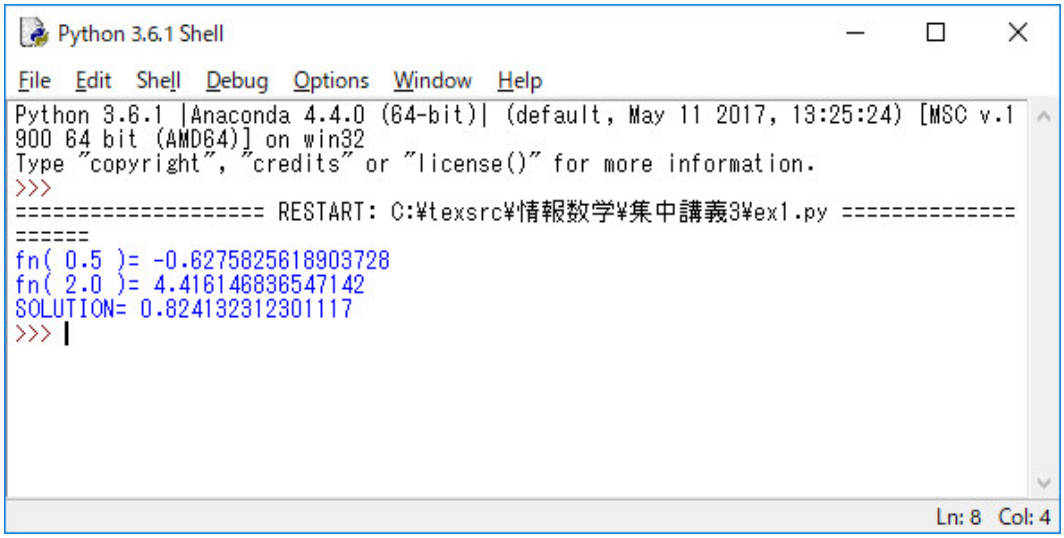
実行します。

次は $f(x) = x^2 - \cos x = 0$ の近似解を求める二分法のプログラムである。二分法ではグラフを描くとか何らかの方法で $f(a) \times f(b) < 0$ となる初期値 a, b を見つけなければならない。

```
import math
```

```
def fn(x) :
    return x * x - math.cos(x)

eps = 0.00000000001
a = 0.5
b = 2.0
m = (a+b)/2
print( "fn(",  a,  ")=", fn(a) )
print( "fn(",  b, ")=",  fn(b) )
while  math.fabs(fn(a)-fn(b)) > eps :
    m = (a+b)/2
    if  fn(m) > 0 :
       b = m
    else :
       a = m
print(  "SOLUTION=", m )
```

実行します。

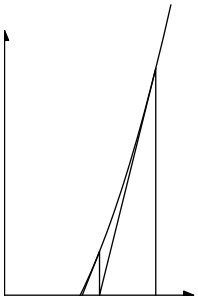


次に **Newton-Raphson** 法で $f(x) = x^2 - \cos x$ の解を求める。

$f(x) = 0$ の解 x の近似解を x_k とし、**Taylor** 級数展開の x に関して一次までの項で $f(x)$ を近似すると、

$$f(x) = f(x_k) + f'(x_k)(x - x_k) = 0$$

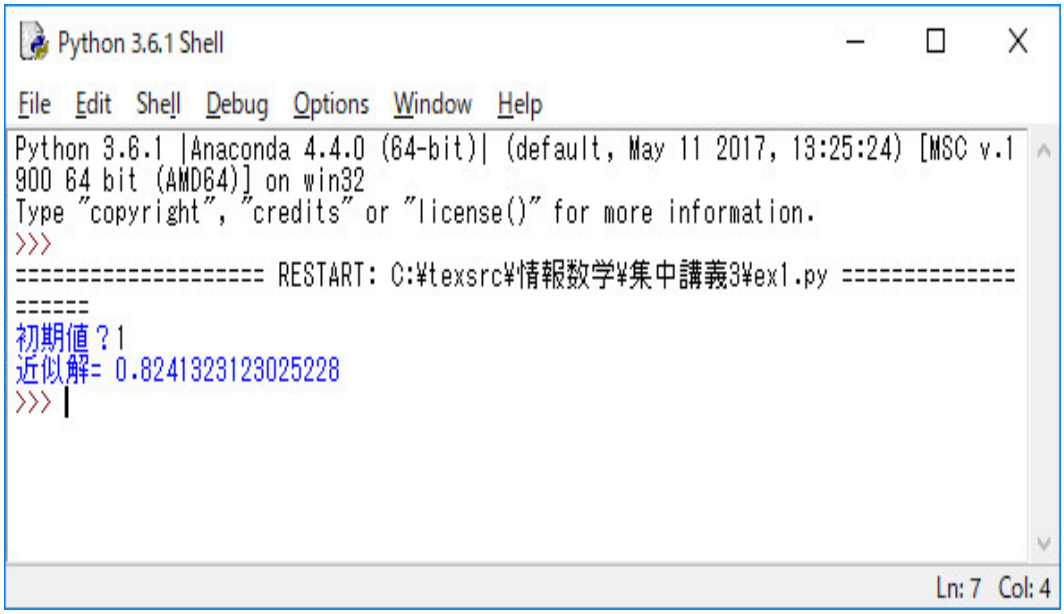
を得る。これより $x_{k+1} = x_k - f(x_k)/f'(x_k)$ の逐次計算を行うと、ある条件のもとで真の解に収束する。



方程式によっては、ニュートン法では、求めたい解の近くに初期値を取っても、別の解の近似値が求まったり、近似解がまったく求められない場合もある。

```
import math

def func(x) :
    return x * x - math.cos(x)

def diff(x) :
    dx = 0.0000001
    return (func(x+dx)-func(x))/dx

max = 100
x = float(input( "初期値？"))
i = 0
while True :
    dx = - func(x) / diff(x)
    x = x + dx;
    if i > max or  math.fabs(dx) < 0.0000001 : break
print(  "近似解=",  x )
```

実行します。



同じことを **Java** でプログラミングすると次のようになる。

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Newton {
    static double f(double x) {
        return x * x - Math.cos(x);
    }
    static double df(double x) {
        return 2 * x + Math.sin(x);
    }
    public static void main(String[] args) {
        int i, max = 100;
        double x, dx;

        BufferedReader rd =
            new BufferedReader(new InputStreamReader(System.in));
        try {
            String line;
            System.out.print("初期値？");
            line = rd.readLine();
            x = Double.parseDouble(line);
        }
        catch(IOException e) {
            System.out.println("入力エラーが発生しました。");
```

```
            return; // 入力エラーなら終了する
        }
        catch(NumberFormatException e) {
            System.out.println("実数を入力して下さい。");
            return; // 入力ミスなら終了する
        }
        System.out.println("  i       x            f(x)         df(x)");
        i=0;
        do {
            dx = - f(x) / df(x);
            x += dx;
            System.out.println(i+","+x+","+f(x)+","+df(x));
            i++;
        } while((i <= max) && (Math.abs(dx) > 0.00000001));
        System.out.println("解="+x);
    }
}
```

実行すると

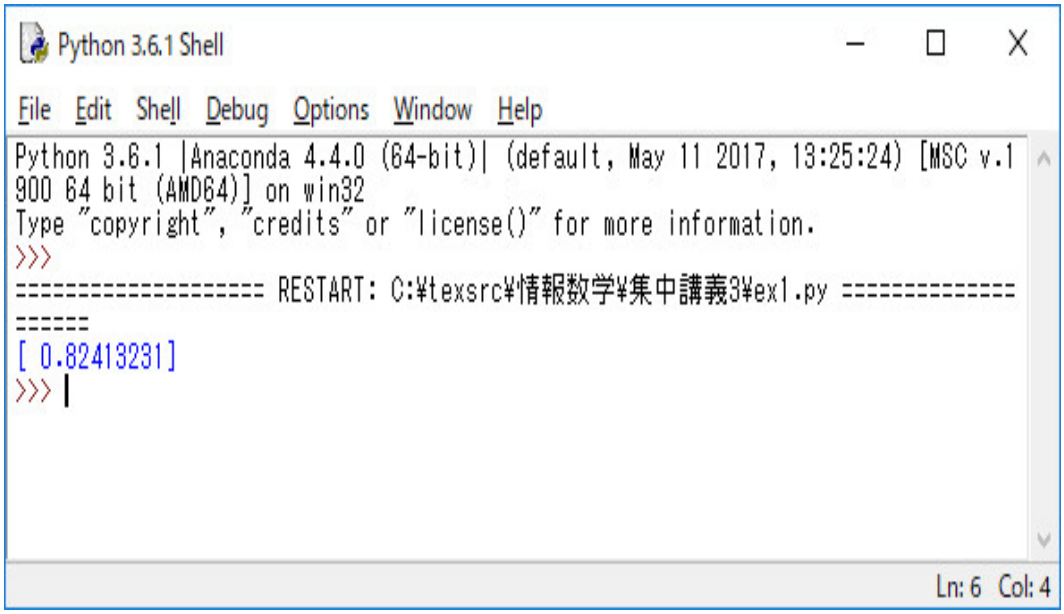
初期値？2

```
  i       x            f(x)         df(x)
0,1.1004523758499203,0.75780251717421,3.0923172164951502
1,0.8553926151203572,0.07577432278402518,2.465613729520581
2,0.8246601764269862,0.0012578634935394017,2.383637503039626
3,0.8241324688825565,3.730086470810079E-7,2.382223774388821
4,0.8241323123025361,3.26405569239796E-14,2.382223354880568
5,0.8241323123025225,2.220446049250313E-16,2.3822233548805314
解=0.8241323123025225
```

となる。SciPy を使うと

```
from scipy import optimize
from math import *

def f(x):
    return x*x - cos(x)

root = optimize.fsolve(f, 1)
print( root )
```

実行すると



です。

注　上の図は

```
prologues := 1;
defaultfont := "rptmr";
defaultscale := 1.5;
beginfig(1);
u=1cm;
vardef f(expr x)=x*x-cosd(x) enddef;
vardef g(expr x)=2*x+sind(x) enddef;
drawarrow (-2.5u,0)--(2.5u,0);
drawarrow (0,-1.1u)--(0,3.5u);
numeric a, b, c;
a := -2.2;
draw (a*u,f(a)*u)
for i=-2.2 step 0.2 until 2.3: ..(i*u, f(i)*u) endfor;
b := 2-f(2)/g(2);
c := b-f(b)/g(b);
draw (2u,0)--(2u,f(2)*u)--(b*u,0)--(b*u,f(b)*u)--(c*u,0);
label.bot(btex $x_0$ etex, (2u,0));
label.bot(btex $x_1$ etex, (b*u,0));
endfig;
end.
```

というプログラムで、 MetaPost を使って作画した。MetaPost は John Hobby が作った METAFONT 類似のソフトで、 PostScript（EPS）形式のファイルを出力します。 1995 年から AT&T ライセンス不要のフリーソフトになりました。METAPOST は METAFONT をお手本にして作られたプログラミング言語で、 METAFONT は白黒の図形しか作れないのに対して、METAPOST なら

フルカラーの図形が作れるなどの細かい違いはありますが、共通している部分が多いです。MetaPost は LaTeX を導入すれば、多分自動的にインストールされている。METAFONT は Donald E. Knuth が作ったフォント作成ツールで、 METAFONT を使って色々な gf 形式のフォントが作れます。 METAFONT を使い、色々の人の無料奉仕で、アラビア語やロシア語やギリシャ語や漢文など色々な言語が LaTeX で使えるようになっています。更に、LaTeX は数式や楽譜や化学式などが綺麗に書ける無料のワープロソフトで、数学の先生は全員使っています。このテキストも LaTeX で作っています。

発展問題：ルート 2 の計算の歴史

$\sqrt{2}$ のもっと上手な計算法　$\sqrt{a+b}$ において、b が a に比べて小さいものと考えると、$\sqrt{a+b} \approx \sqrt{a}$ となります。$\sqrt{a+b}$ と $\sqrt{a}$ との差である未知量 δ を考えて近似を良くしようとすると、

$$\sqrt{a+b} = \sqrt{a} + \delta$$

より

$$a+b = \left(\sqrt{a}+\delta\right)^2 = a + 2\sqrt{a}\delta + \delta^2$$

となります。δ が小さいから、δ^2 を無視すれば、$\delta = b/(2\sqrt{a})$ が得られ、したがって、

$$\sqrt{a+b} \approx \sqrt{a} + \frac{b}{2\sqrt{a}} (|b| \ll |a| \text{ のとき})$$

が得られます。さて、$\sqrt{2}$ の近似値として $v = 1.4$ から始めることにして、$a = v^2, b = 2-a = 2-v^2$ とおきます。すると、上の式より新しい近似値として

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

が得られ、この公式を次々と使うことが出来て、

1.4

1.414285

1.4142135642

1.4142135623730950499

1.41421356237309504880168872420969807 90

が得られます。まったく同じ計算を 60 進法で行うことができ、$1,25$ (即ち $1+\frac{25}{60}$) で始めると $1,24,51,10$ (即ち $1+\frac{24}{60}+\frac{51}{60^2}+\frac{10}{60^3}$) が得られます。この値が紀元前 1900 年のバビロニアの粘土板で見つかるのです。

1450 年頃アルカルサーディーが上の式を改良します。

$$\sqrt{a+b} = \sqrt{a} + \frac{b}{2\sqrt{a}} + \delta$$

を考えて、平方を計算すると、

$$a+b = a+b+\frac{b^2}{4a} + 2\sqrt{a}\delta + \frac{b\delta}{\sqrt{a}} + \delta^2$$

が得られます。最後の 2 項を無視すれば、

$$\sqrt{a+b} \approx \sqrt{a} + \frac{b}{2\sqrt{a}} - \frac{b^2}{8\sqrt{a^3}}$$

が得られます。$\sqrt{2}$ に対して今度は

$$v + \frac{2 - v^2}{2v} - \frac{4 - 4v^2 + v^4}{8v^3} = \frac{3v}{8} + \frac{3}{2v} - \frac{1}{2v^3}$$

が得られ、$v = 1.4$ に次々と使っていけば、速く収束する近似列

1.4

1.4142128

1.41421356237309504870

1.41421356237309504880168872420969807856967187537694807317643

が得られます。上の近似式は $\sqrt{a}$ で割って、$\frac{b}{a}$ を x で置き換えると極めて簡潔な式

$$\sqrt{1+x} \approx 1 + \frac{x}{2}, \sqrt{1+x} \approx 1 + \frac{x}{2} - \frac{x^2}{8}$$

になります。もっと精密な近似式

$$\sqrt{1+x} = 1 + \frac{1}{2}x - \frac{1}{8}x^2 + \frac{1}{16}x^3 - \frac{5}{128}x^4 + \cdots$$

をニュートンが 1665 年に見つけます。

発展問題：複素関数のニュートン法

Newton-Raphson 法は複素関数に対してもまったく同様に使えます。

$f(z) = 0$ の解 z の近似解を z_k とし、Taylor 級数展開の z に関して一次までの項で $f(z)$ を近似すると、

$$f(z) = f(z_k) + f'(z_k)(z - z_k) = 0$$

を得る。これより $z_{k+1} = z_k - f(z_k)/f'(z_k)$ の逐次計算を行うと、ある条件のもとで真の解に収束する。

方程式によっては、ニュートン法では、求めたい解の近くに初期値を取っても、別の解の近似値が求まったり、近似解がまったく求められない場合もある。$f(z) = z^3 - 1$ の根は $z = 1, z = -\frac{1}{2} \pm \frac{\sqrt{3}}{2}i$ です。初期値がどの根に収束するか、色分けすると面白い図ができます。

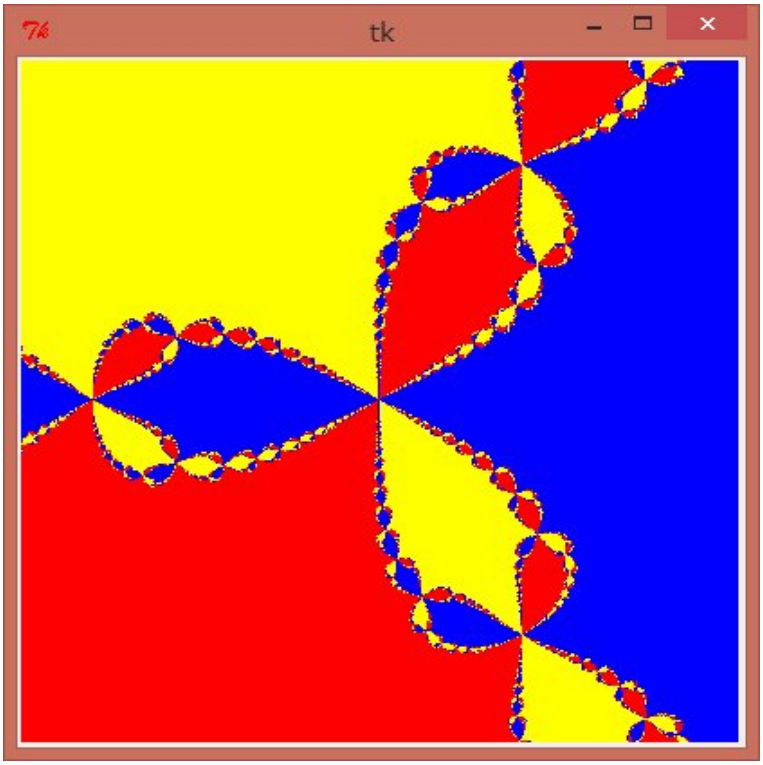
この図を描く Python のプログラムは次のようになります。

```
from tkinter import *
import math

def f(z) :
    return z*z*z-1
def df(z) :
    return 3*z*z

root = Tk()
canvas = Canvas(root, width = 400, height=400)
x0 = y0 = 200
limit = 0.0000001
x = -1.0
while x <= 1 :
```

```
    y = -1.0
    while y <= 1 :
        z = complex(x, y)
        while abs(z-1)>limit and \
              abs(z-complex(-0.5,-math.sqrt(3)/2))>limit and \
              abs(z-complex(-0.5, math.sqrt(3)/2))>limit :
            z = z - f(z) / df(z)
        if abs(z-1) <= limit :
            gx = x0+200*x
            gy = y0-200*y
            canvas.create_line(gx, gy, gx+1, gy+1, fill = 'blue')
        elif abs(z-complex(-0.5,-math.sqrt(3)/2)) <= limit :
            gx = x0+200*x
            gy = y0-200*y
            canvas.create_line(gx, gy, gx+1, gy+1, fill = 'red')
        else:
            gx = x0+200*x
            gy = y0-200*y
            canvas.create_line(gx, gy, gx+1, gy+1, fill = 'yellow')
        y += 0.005
    x += 0.005
canvas.pack()
root.mainloop()
```

Python はデータとして複素数が使えるのでプログラムングが楽です。気になるならインターネットで調べて下さい。

実行すると



となります。

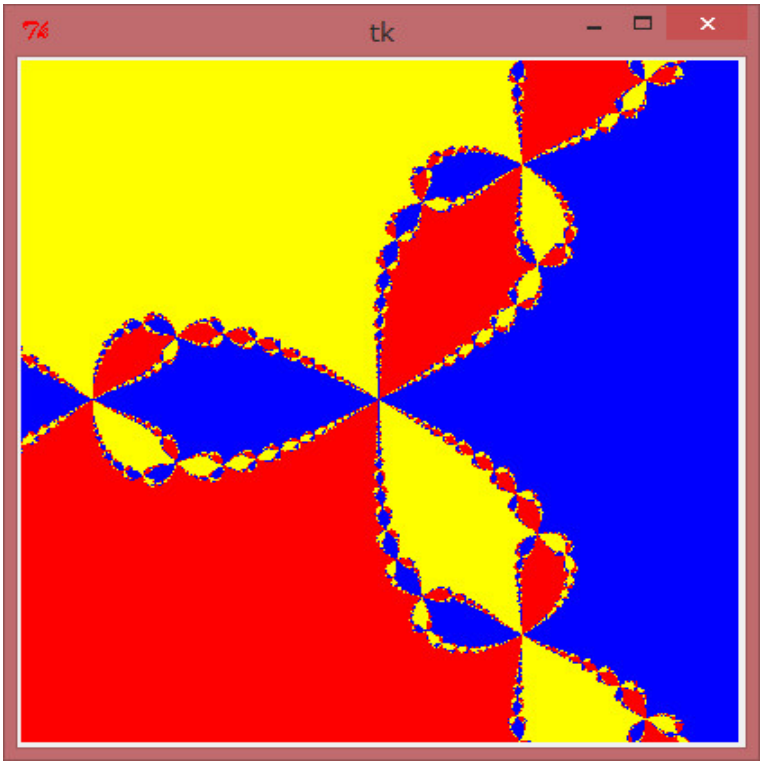
これを作り変えて、マウスで範囲を指定して、拡大するプログラムは

```
from tkinter import *
import math

def f(z) :
    return z*z*z-1
def df(z) :
    return 3*z*z

root = Tk()
canvas = Canvas(root, width = 400, height=400)
sx , sy = 0.0, 0.0
def setsource(event):
    global sx, sy
    sx = (xmax-xmin)/400.0*event.x + xmin
    sy = ymax - (ymax-ymin)/400.0*event.y
canvas.bind("<Button-1>", setsource)
tx, ty = 0.0, 0.0
def drawline(event):
    global sx, sy, tx, ty
    tx = (xmax-xmin)/400.0*event.x + xmin
    ty = ymax - (ymax-ymin)/400.0*event.y
```

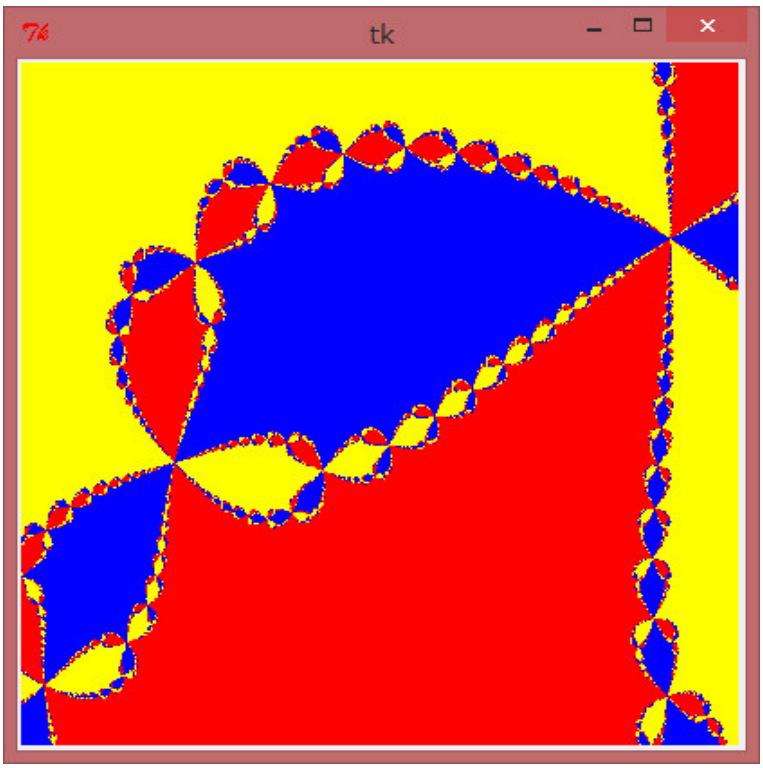
```
    gsx = 400/(xmax-xmin)*(sx-xmin)
    gsy = 400/(ymax-ymin)*(ymax-sy)
    gtx = 400/(xmax-xmin)*(tx-xmin)
    gty = 400/(ymax-ymin)*(ymax-ty)
    canvas.create_rectangle(gsx, gsy, gtx, gty)
canvas.bind("<B1-Motion>", drawline)
def redraw(event):
    global sx, sy, tx, ty
    global xmax, xmin, ymax, ymin
    global dx, dy
    tx = (xmax-xmin)/400.0*event.x + xmin
    ty = ymax - (ymax-ymin)/400.0*event.y
    if tx < sx:
        sx, tx = tx, sx
    if ty < sy:
        sy, ty = ty, sy
    xmin, xmax = sx, tx
    ymin, ymax = sy, ty
    dx = (xmax-xmin)/400.0
    dy = (ymax-ymin)/400.0
    x = xmin
    while x <= xmax :
        y = ymin
        while y <= ymax :
            z = complex(x, y)
            while abs(z-1)>limit and \
                  abs(z-complex(-0.5,-math.sqrt(3)/2))>limit and \
                  abs(z-complex(-0.5, math.sqrt(3)/2))>limit :
                z = z - f(z) / df(z)
            if abs(z-1) <= limit :
                gx = 400/(xmax-xmin)*(x-xmin)
                gy = 400/(ymax-ymin)*(ymax-y)
                canvas.create_line(gx, gy, gx+1, gy+1, fill = 'blue')
            elif abs(z-complex(-0.5,-math.sqrt(3)/2)) <= limit :
                gx = 400/(xmax-xmin)*(x-xmin)
                gy = 400/(ymax-ymin)*(ymax-y)
                canvas.create_line(gx, gy, gx+1, gy+1, fill = 'red')
            else:
                gx = 400/(xmax-xmin)*(x-xmin)
                gy = 400/(ymax-ymin)*(ymax-y)
                canvas.create_line(gx, gy, gx+1, gy+1, fill = 'yellow')
            y += dy
        x += dx
```

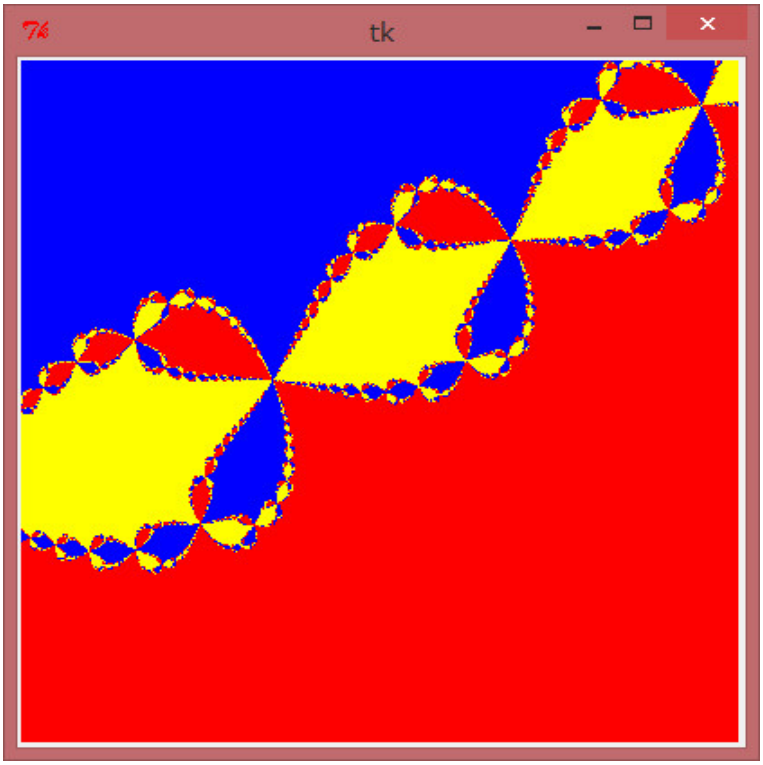
```
canvas.bind("<ButtonRelease-1>", redraw)

x0 = y0 = 200
limit = 0.0000001
xmin, xmax = -1.0, 1.0
ymin, ymax = -1.0, 1.0
x = xmin
while x <= xmax :
    y = ymin
    while y <= ymax :
        z = complex(x, y)
        while abs(z-1)>limit and \
              abs(z-complex(-0.5,-math.sqrt(3)/2))>limit and \
              abs(z-complex(-0.5, math.sqrt(3)/2))>limit :
            z = z - f(z) / df(z)
        if abs(z-1) <= limit :
            gx = 400/(xmax-xmin)*(x-xmin)
            gy = 400/(ymax-ymin)*(ymax-y)
            canvas.create_line(gx, gy, gx+1, gy+1, fill = 'blue')
        elif abs(z-complex(-0.5,-math.sqrt(3)/2)) <= limit :
            gx = 400/(xmax-xmin)*(x-xmin)
            gy = 400/(ymax-ymin)*(ymax-y)
            canvas.create_line(gx, gy, gx+1, gy+1, fill = 'red')
        else:
            gx = 400/(xmax-xmin)*(x-xmin)
            gy = 400/(ymax-ymin)*(ymax-y)
            canvas.create_line(gx, gy, gx+1, gy+1, fill = 'yellow')
        y += 0.005
    x += 0.005
canvas.pack()
root.mainloop()
```

です。実行すると



となります。拡大すると





となります。どこまで拡大しても境界は複雑なままです。

演習問題

1. 立方根の近似値を計算するプログラムを書け。
2. ニュートン法によって 1, 2, ... , 20 の平方根の近似値を計算し、結果を数表の形に見やすく出力するプログラムを書け。
3. 関数 $f(x) = x^2 - \cos x$ のグラフを描くプログラムを書け。

2.3 数値積分

定積分を数値的に求める方法が数学 C で説明されている。区分求積法、数値積分法として台形公式とシンプソンの公式である。区分求積法は定積分の定義であるが、普通は収束が速い台形公式やシンプソンの公式が使われる。$\int_a^b f(x)dx$ の値を近似的に計算することを考える。

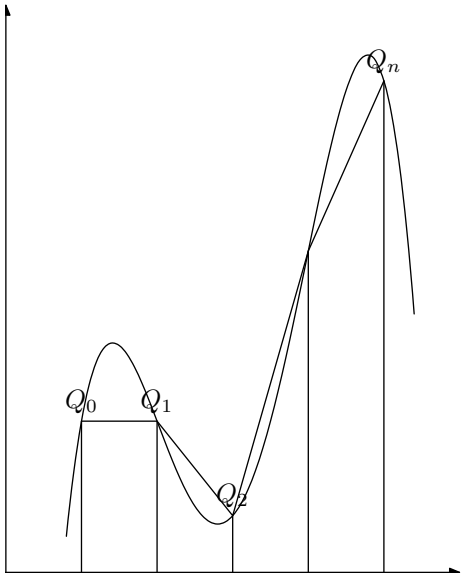
いま、区間 $[a, b]$ を n 等分して、幅が $h = (b-a)/n$ の小区間に分割する。そして、分点を小さい方から順に $P_0, P_1, P_2, \cdots, P_n$ とし、その x 座標の値を $a = x_0, x_1, x_2, \cdots, x_n = b$ とする。さらに、各々の分点 P_k における関数の値を $y_k = f(x_k)$ $(k = 0, 1, 2, \cdots, n)$ とし、座標 (x_k, y_k) の点を Q_k とする。曲線の弧 Q_kQ_{k+1} と三線分 $Q_kP_k, P_kP_{k+1}, P_{k+1}Q_{k+1}$ で囲まれる図形の面積を弧 Q_kQ_{k+1} を線分 Q_kQ_{k+1} で近似し、台形の面積

$$\frac{h}{2}(y_k + y_{k+1})$$

で近似する。

これらを加えると、つぎの公式が得られる。

$$\int_a^b f(x)dx \approx \frac{h}{2}\{y_0 + 2(y_1 + y_2 + \cdots + y_{n-1}) + y_n\}$$



次は台形公式公式による $\int_1^2 \frac{1}{x}dx$ の近似値を求める **BASIC** のプログラムである。

```
100 REM 台形公式
110 A=1:B=2
120 DEF FNY(X)=1/X
130 N=4
140 H=(B-A)/N
150 S=0
160 FOR K=1 TO N-1
170   S=S+FNY(A+K*H)
180 NEXT K
190 S=(FNY(A)+2*S+FNY(B))*H/2
200 PRINT "S=";S
210 PRINT "log(2)=";LOG(2)
220 END
```

実行すると

```
S= 0.6970238095238095
log(2)= 0.6931471805599453
```

です。

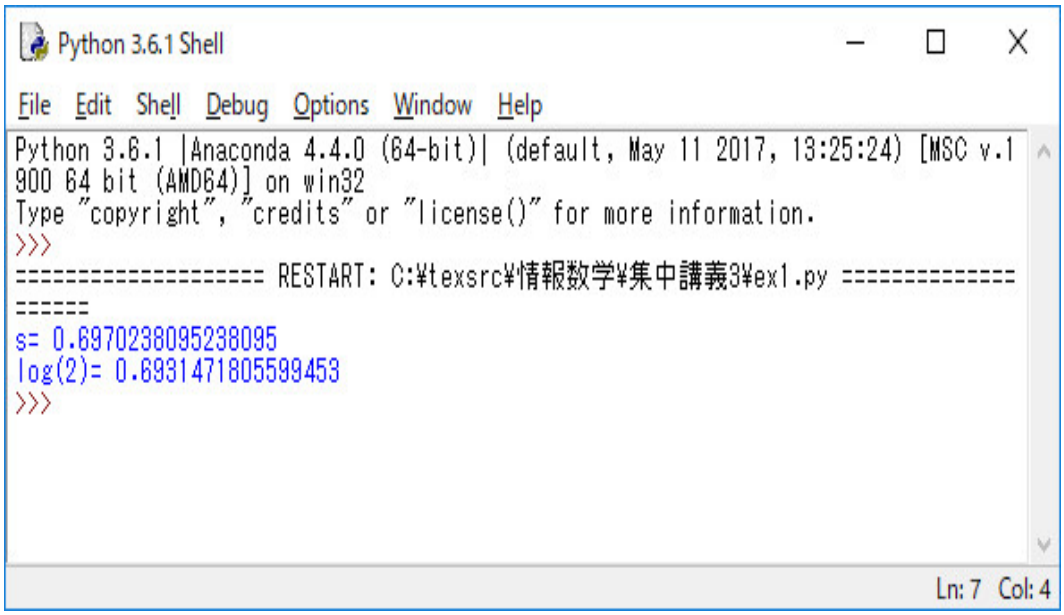
```
210 PRINT "log(2)=";LOG(2)
```

は、真の値と比較するために入れています。分割数 N が小さすぎるので、余り良い近似値ではないです。

上の **BASIC** のプログラムを **Python** に翻訳すると次のようになります。

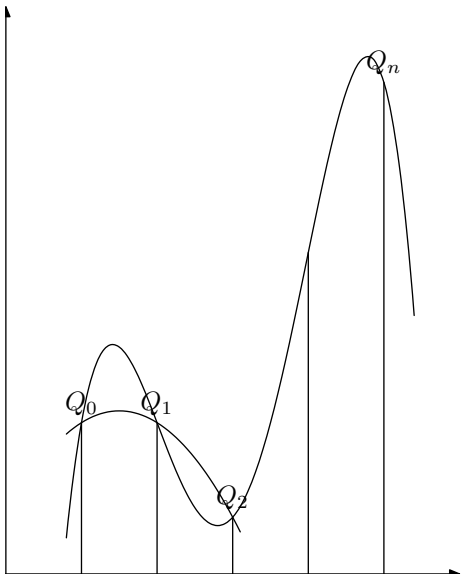
```
import math

def func(x) :
    return 1.0/x

a = 1.0
b = 2.0
n = 4
h = (b-a)/n
s = 0.0
for k in range(1, n) :
    s += func(a+k*h)
s = (func(a)+2*s+func(b))*h/2
print( "s=", s )
print( "log(2)=", math.log(2.0) )
```

実行すると



となります。

シンプソンの公式の公式は次のようにして求める。台形公式で n 等分していたのを今度は $2n$ 等分する。曲線 $y = f(x)$ の弧 $Q_{2k}Q_{2k+1}Q_{2k+2}$ の代わりに点 $Q_{2k}, Q_{2k+1}, Q_{2k+2}$ を通る放物線で近似し面積を求める。



$c = \frac{a+b}{2}$ とし、点 $(a, f(a)), (c, f(c)), (b, f(b))$ を通る放物線を $y = ux^2 + vx + w$ とする。

$$\begin{cases} f(a) = ua^2 + va + w \\ f(b) = ub^2 + vb + w \\ f(c) = uc^2 + vc + ww \end{cases}$$

が成り立つ。したがって、

$$\begin{aligned} w &= f(a) - ua^2 - va \\ v(b-a) &= f(b) - f(a) + u(a^2 - b^2) \\ u &= \frac{2}{(a-b)^2}(f(a) + f(b) - 2f(c)) \end{aligned}$$

が成り立つ。それらを次の式に代入する。

$$\begin{aligned} &\textstyle\int_a^b (ux^2 + vx + w)dx \\ &= (b-a)\{\tfrac{1}{3}u(b^2 + ab + a^2) + \tfrac{1}{2}v(b+a) + w\} \\ &= (b-a)\{\tfrac{1}{3}u(b^2 + ab + b^2) + \tfrac{1}{2}v(b+a) + f(a) - ua^2 - va\} \\ &= (b-a)\{\tfrac{1}{3}u(b^2 + ab - 2a^2) + \tfrac{1}{2}v(b-a) + f(a)\} \\ &= (b-a)\{\tfrac{1}{3}u(b^2 + ab - 2a^2) + \tfrac{1}{2}(f(b) - f(a) + u(a^2 - b^2)) + f(a)\} \\ &= (b-a)\{\tfrac{1}{3}\tfrac{2}{(a-b)^2}(f(a) + f(b) - 2f(c))\tfrac{-a^2+2ab-b^2}{2} + \tfrac{1}{2}(f(b) + f(a))\} \\ &= \tfrac{b-a}{6}(f(b) + f(a) + 4f(c)) \end{aligned}$$

したがって、 $h = (b-a)/(2n)$ とすれば、近似値は

$$\frac{h}{3}(y_{2k+2} + 4y_{2k+1} + y_{2k})$$

である。これらを加えると、次の近似公式がえられる。これをシンプソンの公式という。

$$\int_a^b f(x)dx \approx \frac{h}{3}\{y_0 + 4(y_1 + y_3 + \cdots + y_{2n-1}) + 2(y_2 + y_4 + \cdots + y_{2n-2}) + y_{2n}\}$$

つぎはシンプソンの公式による $\int_1^2 \frac{1}{x}dx$ の近似値を求める **BASIC** のプログラムである。

```
100 REM シンプソンの公式
110 A=1:B=2
120 DEF FNY(X)=1/X
130 N=4
140 H=(B-A)/N
150 S1=0:S2=0
160 FOR K=1 TO N-1 STEP 2
170   S1=S1+FNY(A+K*H):S2=S2+FNY(A+(K+1)*H)
180 NEXT K
190 S=(FNY(A)+4*S1+2*S2-FNY(B))*H/3
200 PRINT "S=";S
210 END
```

実行すると

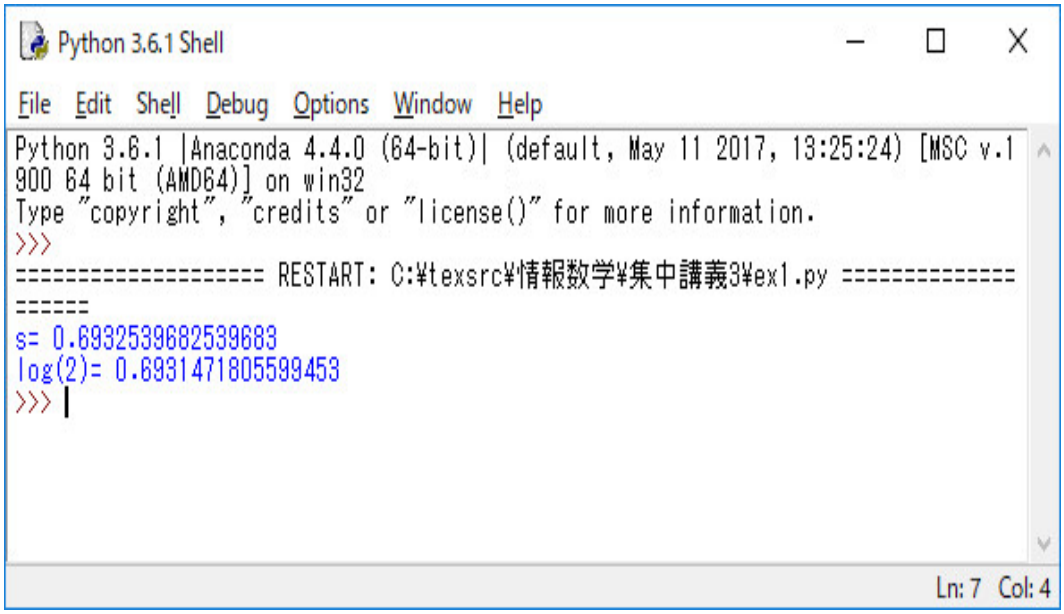
```
S= 0.6931545306545307
```

となる。

上の **BASIC** のプログラムを **Python** に翻訳すると次のようになります。

```
import math

def f(x) :
    return 1.0/x

a = 1.0
b = 2.0
n = 4
h = (b-a)/n
s1 = s2 = 0.0
for k in range(1, n, 2) :
    s1 += f(a+k*h)
    s2 += f(a+(k+1)*h)
s = (f(a) + 4*s1 + 2*s2 - f(b))*h/3
print( "s=", s )
print( "log(2)=", math.log(2.0) )
```

実行すると



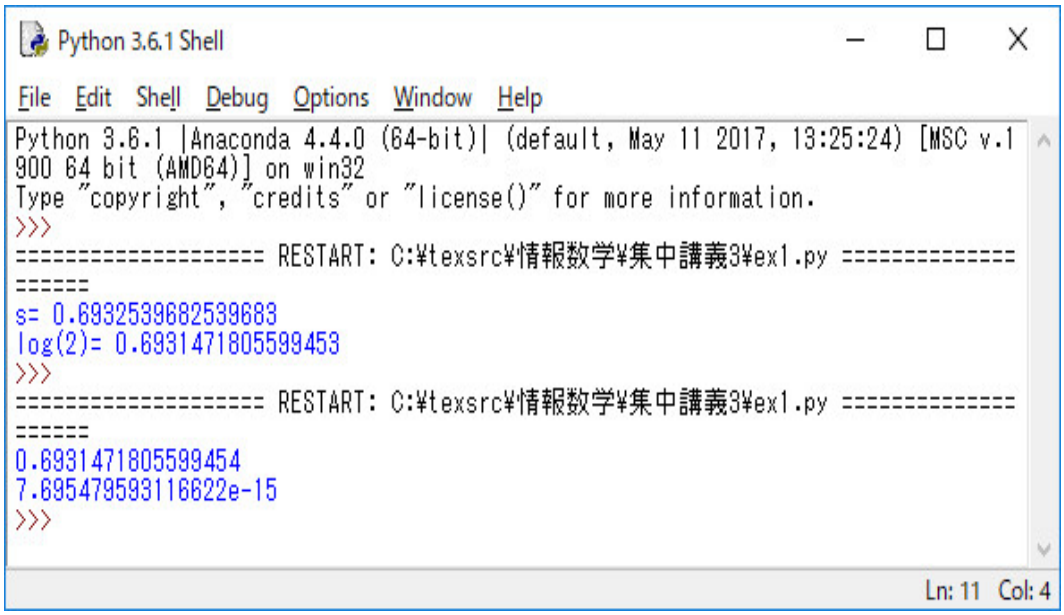
となります。

n の値を大きくすると良い近似が得られます。

Scipy を使うと次のようになります。

```
from scipy.integrate import quad

y, abserr = quad(lambda x: 1/x, 1, 2)
print( y )
print( abserr )
```

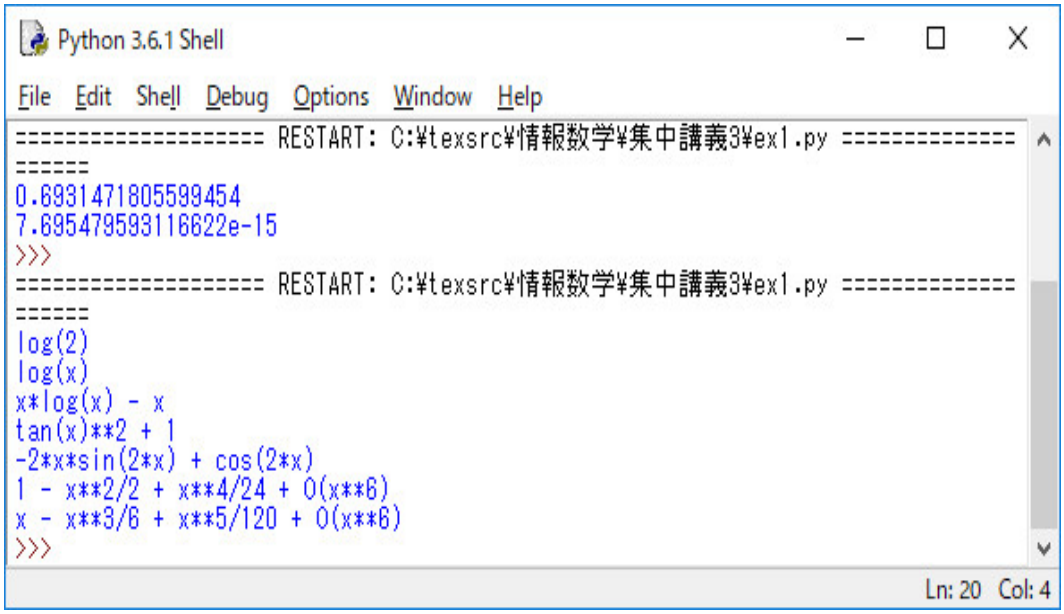
実行すると



となります。下に表示されているのは誤差です。
　更に、SymPy を使うと次のように数式処理ができます。

```
from sympy import *

x = Symbol('x')

print( integrate(1/x, (x, 1, 2)) )
print( integrate(1/x, x) )
print( integrate(log(x), x) )
print( diff(tan(x), x) )
print( diff(x*cos(2*x), x))
print( series(cos(x), x))
print( series(sin(x), x))
```

実行すると



となります。

```
print( integrate(1/x, (x, 1, 2)))
```

は定積分を計算しています。

```
print( integrate(1/x, x))
print( integrate(log(x), x))
```

は不定積分を計算しています。

```
print( diff(tan(x), x))
print( diff(x*cos(2*x), x))
```

は微分を計算しています。

```
print( series(cos(x), x))
print( series(sin(x), x))
```

は Tayler 級数展開を計算しています。

NumPy や SciPy や SymPy のインストール方法や使い方など興味があれば自分でインターネットで調べてください。

演習問題

1. 定積分 $\int_0^1 \sqrt{1-x^2}dx$ を、区間の分割数が１０、２０、…、６０の場合についてシンプソンの公式で計算し、誤差を調べるプログラムを書け。ただし、この定積分の値は $\frac{\pi}{4}$ であり、有効桁数６桁で表すと０．７８５３９６である。

2. 放物線 $y = x^2$ の $x = 0$ から $x = 1$ までの弧長を求めるプログラムを書け。

2.4 連立一次方程式の解法

数学 **C** では連立一次方程式を解くガウスの消去法が説明され **BASIC** のプログラムが与えられている。つぎの **BASIC** のプログラムはガウスの消去法で連立一次方程式を解くものである。

```
100 REM ガウスの消去法
110 INPUT "N=";N
120 DIM A(N,N),B(N)
130 REM 方程式の入力
140 FOR I=1 TO N
150   FOR J=1 TO N
160     PRINT "A(";USING "#" I;USING "#" J;")=";:INPUT A(I,J)
170   NEXT J
180   PRINT "B(";USING "#" I;")=";:INPUT B(I)
190 NEXT I
200 REM 前進消去
210 FOR K=1 TO N-1
220   REM 行の入れ換え
230   AMAX=ABS(A(K,K)):KMAX=K
240   FOR I=K+1 TO N
250     IF ABS(A(I,K))>AMAX THEN KMAX=I:AMAX=ABS(A(I,K))
260   NEXT I
270   IF AMAX<0.00001 THEN PRINT "解けない":END
280   IF KMAX=K THEN 340
290   FOR J=K TO N
300     T=A(K,J):A(K,J)=A(KMAX,J):A(KMAX,J)=T
310   NEXT J
320   T=B(K):B(K)=B(KMAX):B(KMAX)=T
330   REM 前進消去の中心部
```

```
340   FOR I=K+1 TO N
350     A(I,K)=A(I,K)/A(K,K)
360     FOR J=K+1 TO N
370       A(I,J)=a(I,J)-A(I,K)*A(K,J)
380     NEXT J
390     B(I)=B(I)-A(I,K)*B(K)
400   NEXT I
410 NEXT K
420 IF ABS(A(N,N))<0.00001 THEN PRINT "解けない" : END
430 REM 後退代入
440 FOR K=N TO 1 STEP -1
450   FOR I=K+1 TO N
460     B(K)=B(K)-A(K,I)*B(I)
470   NEXT
480   B(K)=B(K)/A(K,K)
490 NEXT K
500 REM 解の出力
510 PRINT "X ";
520 FOR K=1 TO N
530   PRINT B(K);
540 NEXT K
550 END
```

実行すると

```
N=3
A( 1 1 )= ? 1
A( 1 2 )= ? 2
A( 1 3 )= ? -1
B( 1 )= ? -3
A( 2 1 )= ? 2
A( 2 2 )= ? 4
A( 2 3 )= ? 1
B( 2 )= ? 0
A( 3 1 )= ? 3
A( 3 2 )= ? 8
A( 3 3 )= ? 1
B( 3 )= ? -3
X  1.0000000000000007 -1.0000000000000002 2.0
```

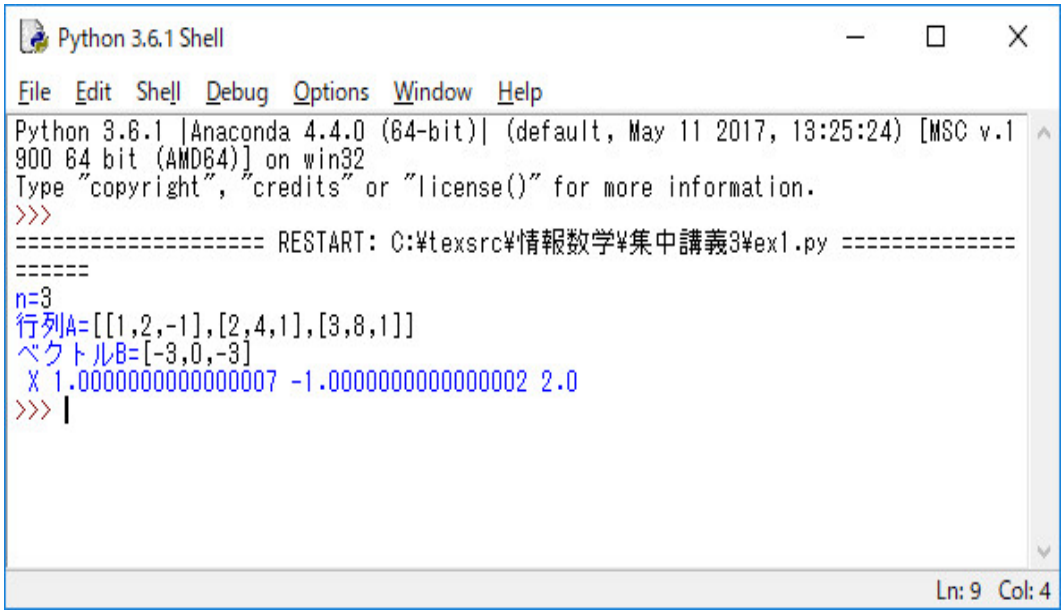
となる。

上の **BASIC** のプログラムを **Python** に翻訳すると次のようになります。

```
import math
import sys
```

```
n = int(input("n="))
A = eval(input("行列 A="))
B = eval(input("ベクトル B="))
for k in range(n) :
    AMAX = math.fabs(A[k][k])
    KMAX = k
    for i in range(k+1, n) :
        if math.fabs(A[i][k]) > AMAX :
            KMAX = i
            AMAX = math.fabs(A[i][k])
    if AMAX < 0.00001 :
        print(  "解けない" )
        sys.exit()
    if KMAX != k :
        for j in range(k, n) :
            T = A[k][j]
            A[k][j] = A[KMAX][j]
            A[KMAX][j] = T
        T = B[k]
        B[k] = B[KMAX]
        B[KMAX] = T
    for i in range(k+1, n) :
        A[i][k] = A[i][k] / A[k][k]
        for j in range(k+1, n) :
            A[i][j] = A[i][j] - A[i][k] * A[k][j]
        B[i] = B[i] - A[i][k] * B[k]
if math.fabs(A[n-1][n-1]) < 0.00001 :
    print( "解けない" )
    sys.exit()
l = range(n)
l.reverse()
for k in l :
    for i in range(k+1, n) :
        B[k] = B[k] - A[k][i] * B[i]
    B[k] = B[k] / A[k][k]
print( " X", end=' ')
for k in range(n) :
    print( B[k], end=" ")
```

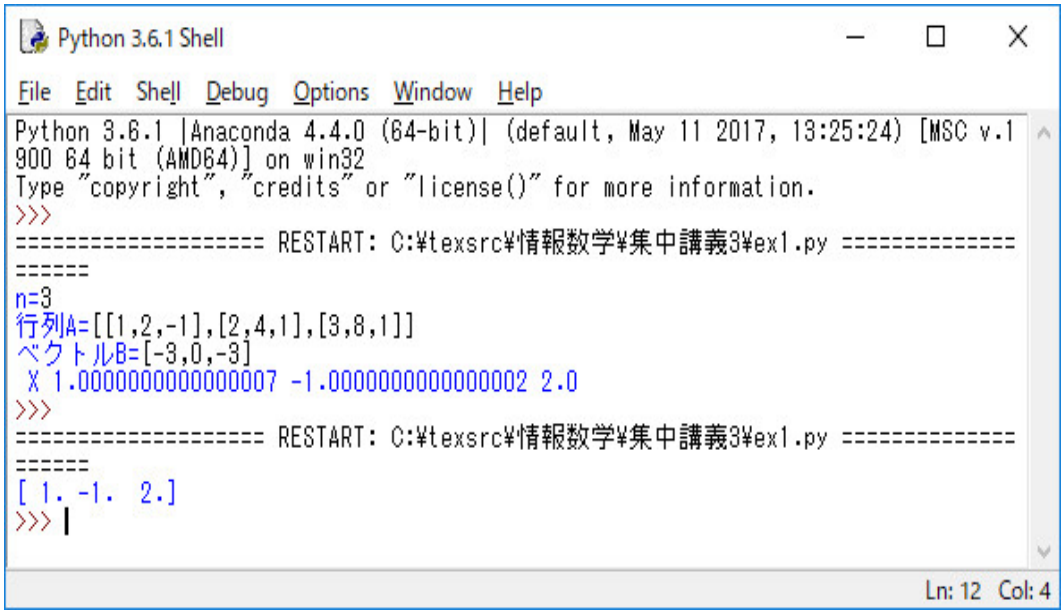
実行すると



となります。これは NumPy を使えば、次のようにすればよい。

```
import numpy as np

A = np.array([[1, 2, -1], [2, 4 , 1], [3, 8, 1]])
b = np.array([-3, 0, -3])
x = np.linalg.solve(A, b)
print( x )
```

実行すると



となります。

n 次正方行列の逆行列を求めるには連立一次方程式を n 個同時に解けばよい。次は逆行列を求める **BASIC** プログラムである。

```
100 REM ガウスの消去法
110 INPUT "N=";N
120 DIM A(N,N),B(N,N)
130 REM 行列の入力
140 FOR I=1 TO N
150   FOR J=1 TO N
160     PRINT "A(";USING "#" I;USING "#" J;")=";:INPUT A(I,J)
170   IF I=J THEN B(I,J)=1 ELSE B(I,J)=0
180   NEXT J
190 NEXT I
200 REM 前進消去
210 FOR K=1 TO N-1
220   REM 行の入れ換え
230   AMAX=ABS(A(K,K)):KMAX=K
240   FOR I=K+1 TO N
250     IF ABS(A(I,K))>AMAX THEN KMAX=I:AMAX=ABS(A(I,K))
260   NEXT I
270   IF AMAX<0.00001 THEN PRINT "解けない":END
280   IF KMAX=K THEN 340
290   FOR J=K TO N
300     T=A(K,J):A(K,J)=A(KMAX,J):A(KMAX,J)=T
310   NEXT J
317   FOR J=1 TO N
320     T=B(K,J):B(K,J)=B(KMAX,J):B(KMAX,J)=T
323   NEXT J
330   REM 前進消去の中心部
340   FOR I=K+1 TO N
350     A(I,K)=A(I,K)/A(K,K)
360     FOR J=K+1 TO N
370       A(I,J)=a(I,J)-A(I,K)*A(K,J)
380     NEXT J
387     FOR J=1 TO N
390       B(I,J)=B(I,J)-A(I,K)*B(K,J)
393     NEXT J
400   NEXT I
410 NEXT K
420 IF ABS(A(N,N))<0.00001 THEN PRINT "解けない" : END
430 REM 後退代入
440 FOR K=N TO 1 STEP -1
450   FOR I=K+1 TO N
```

```
457     FOR J=1 TO N
460       B(K,J)=B(K,J)-A(K,I)*B(I,J)
463     NEXT J
470   NEXT
477   FOR J=1 TO N
480     B(K,J)=B(K,J)/A(K,K)
483   NEXT J
490 NEXT K
500 REM 解の出力
510 PRINT "逆行列 "
520 FOR K=1 TO N
527   FOR J=1 TO N
530     PRINT B(K,J);
533   NEXT J
536   PRINT
540 NEXT K
550 END
```

実行すると

```
N=2
A( 1 1 )= ? 2
A( 1 2 )= ? 1
A( 2 1 )= ? 3
A( 2 2 )= ? 4
逆行列
0.8000000000000002 -0.20000000000000004
-0.6000000000000001 0.4
```

となる。

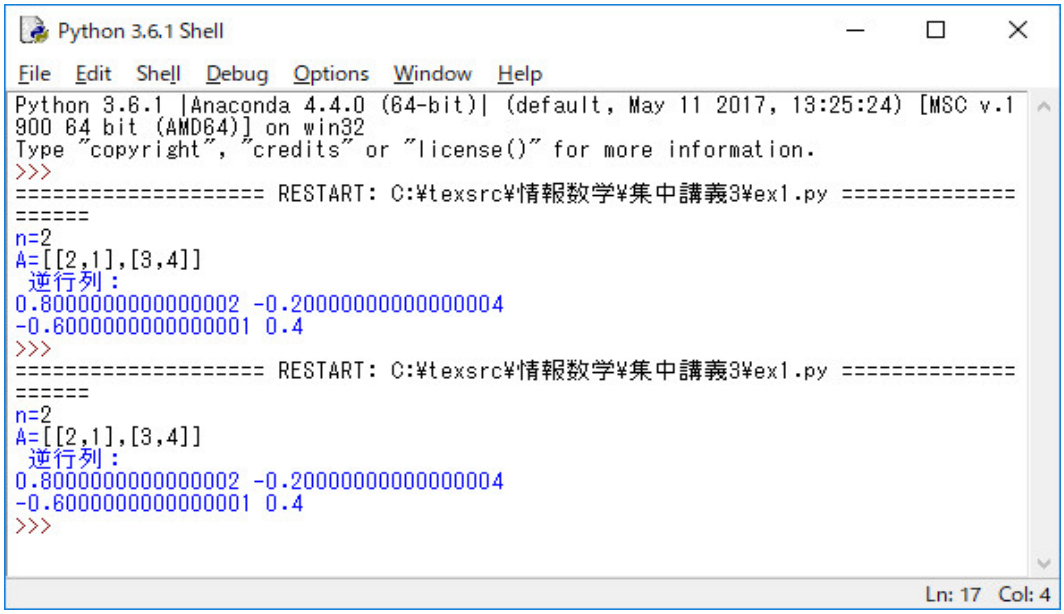
上の **BASIC** のプログラムを **Python** に翻訳すると次のようになります。

```
import math
import sys

n = int(input("n="))
A = eval(input("A="))

B = []
for i in range(n) :
    C = []
    for k in range(n) :
        if k == i :
            C.append(1.0)
        else :
```

```
            C.append(0.0)
    B.append(C)
#print( B )
for k in range(n) :
    AMAX = math.fabs(A[k][k])
    KMAX = k
    for i in range(k+1, n) :
        if math.fabs(A[i][k]) > AMAX :
            KMAX = i
            AMAX = math.fabs(A[i][k])
    if AMAX < 0.00001 :
        print(  "解けない" )
        sys.exit()
    if KMAX != k :
        for j in range(k, n) :
            T = A[k][j]
            A[k][j] = A[KMAX][j]
            A[KMAX][j] = T
        for j in range(n) :
            T = B[k][j]
            B[k][j] = B[KMAX][j]
            B[KMAX][j] = T
    for i in range(k+1, n) :
        A[i][k] = A[i][k] / A[k][k]
        for j in range(k+1, n) :
            A[i][j] = A[i][j] - A[i][k] * A[k][j]
        for j in range(n) :
            B[i][j] = B[i][j] - A[i][k] * B[k][j]
if math.fabs(A[n-1][n-1]) < 0.00001 :
    print ( "解けない" )
    sys.exit()
l = list(range(n))
l.reverse()
for k in l :
    for i in range(k+1, n) :
        for j in range(n) :
            B[k][j] = B[k][j] - A[k][i] * B[i][j]
    for j in range(n) :
        B[k][j] = B[k][j] / A[k][k]
print(  " 逆行列 : ")
for k in range(n) :
    for j in range(n) :
        print( B[k][j], end=" ")
```

```
    print()
```

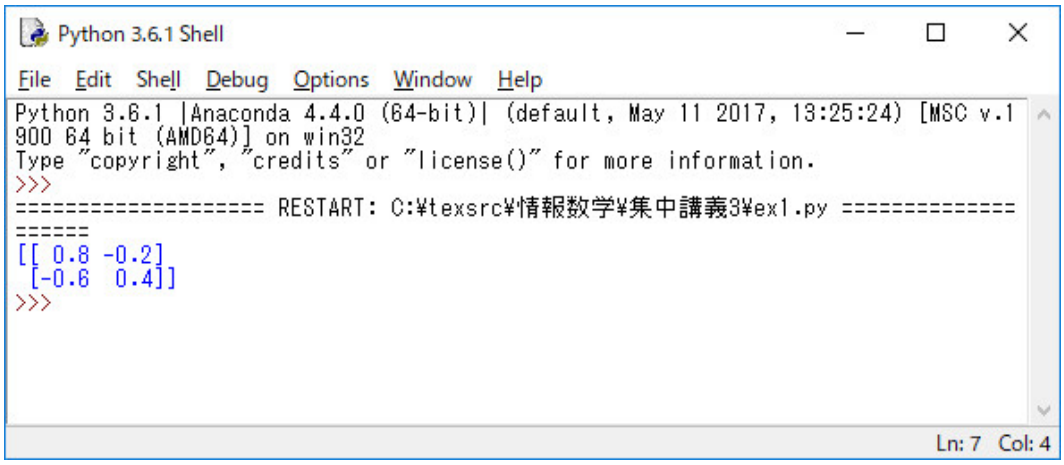
実行すると



となります。これは NumPy を使えば、次のようにすればよい。

```
import numpy as np

A = np.mat("2 1; 3 4")
x = np.linalg.inv(A)
print( x )
```

実行すると



となります。

これらのアルゴリズムの解説は線形代数の教科書を見て下さい。

演習問題

1. 基本的な考え方はガウスの消去法と同じであるが、前進消去の段階で、対角線より下にある成分だけでなく、対角線より上にある成分も同時に消去してしまう方法もある。これをガウス・ジョルダンの消去法あるいは掃出し法という。ガウス・ジョルダンの消去法により連立一次方程式を解くプログラムを書け。

3　オブジェクト指向プログラミング

Python はオブジェクト指向プログラミング言語です。最後にオブジェクト指向プログラミングに挑戦してみましょう。

3.1　行列

今まで行列を [[1,2,3],[4,5,6],[7,8,9]] のようなリストのリストとして考えてきましたが、単にその行列式を計算したり、連立方程式の係数を表現するためにのみ使っていたので、このような表現でよかったのですが、行列の和や積や転置行列を考えるにはリストのリストによる表現では不便です。このような時、Python では行列のためのクラスを作ります。行列の和や積や転置行列ができるクラスを作ってみましょう。

最初のプログラムは

```
def plain_mat(l):
    ls = []
    for e in l:
        ls.extend(e)
    return (ls, len(l[0]), len(l))
class Matrix:
    def __init__(self, b, n, m):
        self.mat = b[:]
        self.w = n
        self.h = m
    def print(self):
        for y in range(self.h):
            for x in range(self.w):
                print( self.mat[(self.w)*y+x],end=' ')
            print()
        print()
if __name__ == '__main__':
    l = eval(input('A='))
    ls, n, m = plain_mat(l)
    A = Matrix(ls, n, m)
    A.print()
```

です。

```
class Matrix:
    def __init__(self, b, n, m):
        self.mat = b[:]
        self.w = n
        self.h = m
    def print(self):
        for y in range(self.h):
            for x in range(self.w):
                print( self.mat[(self.w)*y+x],end=' ')
            print()
        print()
```

が行列のクラス Matrix を定義している部分です。クラスを定義するには

```
class Matrix:
```

のように

```
class クラス名 :
```

から始めます。クラス Matrix で定義されている最初の関数

```
    def __init__(self, b, n, m):
        self.mat = b[:]
        self.w = n
        self.h = m
```

はコンストラクターと呼ばれるメソッドです。クラスに付随する関数はメソッドと呼ばれます。コンストラクターは

```
    A = Matrix(ls, n, m)
```

のように使われ、クラス Matrix のオブジェクト（クラスを体現した実体で、クラスはその設計図です）を作ります。

```
    def __init__(self, b, n, m):
```

の最初の引数は常に self で、メソッドを定義する時の決まり事です。引数 b には行列の行を一列に並べたリストをセットすることを想定し、n には行列の幅（列の個数)、m には行列の高さ（行の個数）がセットされることを想定しています。

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

という行列なら、b = [1,2,3,4,5,6], n = 3, m = 2 です。

```
        self.mat = b[:]
        self.w = n
        self.h = m
```

では、クラス Matrix のメンバー self.mat, self.w, self.h にそれぞれ b, n, m がセットされています。b はリストですから、単なる代入ではなく、コピーされて代入されています。b は単純なリストですから、浅いコピーの self.mat = b[:] が使われています。クラスの定義の中では、クラスのメンバーは必ず self. をメンバー名の頭につけます。各行列を行列の幅と高さと行列の実体を表すリストで保存するようにしようというわけです。

これで行列のオブジェクトが作れるようになりましたが、これだけでは何の役にも立ちません。オブジェクトの内容を表示できるようにするために

```
    def print(self):
        for y in range(self.h):
            for x in range(self.w):
                print( self.mat[(self.w)*y+x],end=' ')
            print()
        print()
```

で、print() メソッドを定義しています。クラスのメンバー関数ですから、引数に self を付けます。オブジェクトのメンバー self.mat の内容を整形してみ易く出力します。

```
    A.print()
```

のように、オブジェクト.print() として使います。

plain_mat(l) 関数

```
def plain_mat(l):
    ls = []
    for e in l:
        ls.extend(e)
    return (ls, len(l[0]), len(l))
```

は、[[1,2,3],[4,5,6]] の形の行列表示を [1,2,3,4,5,6], width=3, hight = 2 という表示に変更して返す関数です。

```
def plain_mat(l):
        ls.extend(e)
```

の extend(e) メソッドはリスト ls の後半に、リスト e の要素を並べる関数です。これで [[1,2,3],[4,5,6]] の形のリストを [1,2,3,4,5,6] の形のリストに変換しています。

最後に

```
    return (ls, len(l[0]), len(l))
```

と ls, len(l[0]), len(l) のタプル (ls, len(l[0]), len(l)) を返しています。これは

```
    ls, n, m = plain_mat(l)
```

の形式で受け取ることが出来ます。最後に

```
if __name__ == '__main__':
    l = eval(input('A='))
    ls, n, m = plain_mat(l)
    A = Matrix(ls, n, m)
    A.print()
```

の部分で、実際にこのクラスを使っています。

```
if __name__ == '__main__':
```

は、上で定義したクラスを別のプログラムで使いたくて、import このプログラムのファイル名 として使うときは、これ以降を無視しなさいと言う指示です。

```
    l = eval(input('A='))
    ls, n, m = plain_mat(l)
    A = Matrix(ls, n, m)
    A.print()
```

では、まず今まで通り [[1,2,3],[4,5,6]] の形でキーボード入力し、Matrix のコンストラクターに渡せるように、`plain_mat(l)` 関数で変形し、A という行列のオブジェクト（実体）を作り、A を表示しています。

このプログラムを実行すると

```
A=[[1,2,3],[4,5,6]]
1 2 3
4 5 6
```

となります。

行列の足し算ができるようにしましょう。

```
def add(A, B):
    return Matrix([d1+d2 for d1, d2 in zip(A.mat, B.mat)], A.w, A.h)
```

を追加します。後半も少し修正して、全体のプログラムは

```
def add(A, B):
    return Matrix([d1+d2 for d1, d2 in zip(A.mat, B.mat)], A.w, A.h)
def plain_mat(l):
    ls = []
    for e in l:
        ls.extend(e)
    return (ls, len(l[0]), len(l))
class Matrix:
    def __init__(self, b, n, m):
        self.mat = b[:]
        self.w = n
        self.h = m
    def print(self):
        for y in range(self.h):
            for x in range(self.w):
                print( self.mat[(self.w)*y+x],end=' ')
            print()
        print()
```

```
if __name__ == '__main__':
    l = eval(input('A='))
    ls, n, m = plain_mat(l)
    A = Matrix(ls, n, m)
    l = eval(input('B='))
    ls, n, m = plain_mat(l)
    B = Matrix(ls, n, m)
    A.print()
    B.print()
    C = add(A, B)
    C.print()
```

となります。add(A, B) の定義 def add(A, B): は後の都合のため、Matrix クラスの定義の前に打ち込んでください。実行すると

```
A=[[1,1,1,1],[1,0,1,0]]
B=[[1,2,3,4],[5,6,7,8]]
1 1 1 1
1 0 1 0

1 2 3 4
5 6 7 8

2 3 4 5
6 6 8 8
```

となります。

```
def add(A, B):
    return Matrix([d1+d2 for d1, d2 in zip(A.mat, B.mat)], A.w, A.h)
```

では、リストの内包表記を使って、行列の和を計算し、結果を Matrix のオブジェクトとして返しています。

```
    C = add(A, B)
```

では、行列の和を計算している雰囲気が出ません。

```
    C = A + B
```

とできるようにしましょう。これは簡単です。

```
class Matrix:
    def __init__(self, b, n, m):
        self.mat = b[:]
        self.w = n
        self.h = m
    __add__ = add
```

```
    def print(self):
        for y in range(self.h):
            for x in range(self.w):
                print( self.mat[(self.w)*y+x],end=' ')
            print()
        print()
```

と、Matrix のクラスの定義に一行

```
    __add__ = add
```

を追加するだけで実現できます。これは演算子のオーバーロードと言います。
引き算もできるようにするためには、次のようにします。

```
def neg(A):
    return Matrix([-d for d in A.mat], A.w, A.h)
```

という関数 neg(A) を定義し、Matrix のクラスの定義を

```
class Matrix:
    def __init__(self, b, n, m):
        self.mat = b[:]
        self.w = n
        self.h = m
    __add__ = add
    __neg__ = neg
    def __sub__(a, b):
        return a+(-b)

    def print(self):
        for y in range(self.h):
            for x in range(self.w):
                print( self.mat[(self.w)*y+x],end=' ')
            print()
        print()
```

に修正します。

```
    __neg__ = neg
    def __sub__(a, b):
        return a+(-b)
```

が追加されました。

```
if __name__ == '__main__':
    l = eval(input('A='))
    ls, n, m = plain_mat(l)
```

```
    A = Matrix(ls, n, m)
    l = eval(input('B='))
    ls, n, m = plain_mat(l)
    B = Matrix(ls, n, m)
    A.print()
    B.print()
    C = A - B
    C.print()
    D = -C
    D.print()
```

を実行すると

```
A=[[1,2,3],[4,5,6]]
B=[[1,1,1],[2,2,2]]
1 2 3
4 5 6

1 1 1
2 2 2

0 1 2
2 3 4

0 -1 -2
-2 -3 -4
```

になります。

行列の掛け算を定義してみましょう。関数 mul(A,B) を行列の積の定義通り

```
def mul(A,B):
    c = []
    for i in range(A.h):
        for j in range(B.w):
            x = 0
            for k in range(A.h):
                x += A.mat[i*A.w+k]*B.mat[k*B.w+j]
            c.append(x)
    return Matrix(c, B.w, A.h)
```

と定義し、

```
    __mul__ = mul
```

を、Matrix クラスの定義に追加すれば、

```
if __name__ == '__main__':
```

```
    l = eval(input('A='))
    ls, n, m = plain_mat(l)
    A = Matrix(ls, n, m)
    l = eval(input('B='))
    ls, n, m = plain_mat(l)
    B = Matrix(ls, n, m)
    A.print()
    B.print()
    C = mul(A,B)
    C.print()
    D = A * B
    D.print()
```

を実行すれば、

```
A=[[1,2],[3,4]]
B=[[0,1,2],[1,0,2]]
1 2
3 4

0 1 2
1 0 2

2 1 6
4 3 14

2 1 6
4 3 14
```

となります。最後に転置行列を与える関数を定義してみます。

```
    def T(self):
        t = []
        for j in range(self.w):
            for i in range(self.h):
                t.append(self.mat[i*self.w+j])
        return Matrix(t, self.h, self.w)
```

を Matrix クラスの定義に追加します。これも定義通りプログラミングしているだけです。全体のプログラムは

```
def add(A, B):
    return Matrix([d1+d2 for d1, d2 in zip(A.mat, B.mat)], A.w, A.h)
def neg(A):
    return Matrix([-d for d in A.mat], A.w, A.h)
def mul(A,B):
```

```
    c = []
    for i in range(A.h):
        for j in range(B.w):
            x = 0
            for k in range(A.h):
                x += A.mat[i*A.w+k]*B.mat[k*B.w+j]
            c.append(x)
    return Matrix(c, B.w, A.h)

def plain_mat(l):
    ls = []
    for e in l:
        ls.extend(e)
    return (ls, len(l[0]), len(l))

class Matrix:
    def __init__(self, b, n, m):
        self.mat = b[:]
        self.w = n
        self.h = m
    __add__ = add
    __neg__ = neg
    def __sub__(a, b):
        return a+(-b)
    __mul__ = mul

    def T(self):
        t = []
        for j in range(self.w):
            for i in range(self.h):
                t.append(self.mat[i*self.w+j])
        return Matrix(t, self.h, self.w)

    def print(self):
        for y in range(self.h):
            for x in range(self.w):
                print( self.mat[(self.w)*y+x],end=' ')
            print()
        print()

if __name__ == '__main__':
    l = eval(input('A='))
    ls, n, m = plain_mat(l)
```

```
    A = Matrix(ls, n, m)
    A.print()
    B = A.T()
    B.print()
```

となりました。実行すると

```
A=[[1,2,3],[4,5,6]]
1 2 3
4 5 6


1 4
2 5
3 6
```

となります。

3.2 クロス・クラム

最後にボードゲームのクロス・クラムを作ってみましょう。先手後手どちらが必勝か小さい盤以外は未解決のゲームです。最終的に作るプログラムをまず提示しておきます。

```
FIRST_PLAYER = 0
SECOND_PLAYER = 1

class Board:
    def __init__(self, b, n, m):
        self.board = b[:]
        self.width = n
        self.hight = m
    def put_piece(self, turn, pos):
        if turn == FIRST_PLAYER:
            self.board[pos] = 1
            self.board[pos+1] = 1
        else:
            self.board[pos] = 2
            self.board[pos+self.width+1] = 2
    def regalmoveP(self, turn, pos):
        if turn == FIRST_PLAYER:
            if self.board[pos] == 0 and self.board[pos+1] == 0:
                return True
            else:
                return False
        else:
```

```
            if self.board[pos] == 0 and self.board[pos+self.width+1] == 0:
                return True
            else:
                return False
    def print_board(self):
        for y in range(self.hight):
            for x in range(self.width):
                print( self.board[(self.width+1)*y+x],end=' ')
            print()
        print()
    def get_candidate(self, turn):
        cand = []
        if turn == FIRST_PLAYER:
            for x in range(self.width-1):
                for y in range(self.hight):
                    if (self.regalmoveP(FIRST_PLAYER, (self.width+1)*y+x)):
                        cand.append((self.width+1)*y+x)
        else:
            for x in range(self.width):
                for y in range(self.hight-1):
                    if (self.regalmoveP(SECOND_PLAYER, (self.width+1)*y+x)):
                        cand.append((self.width+1)*y+x)
        return cand
    def gameoverP(self, turn):
        cand = self.get_candidate(turn)
        if len(cand) == 0:
            return True
        else:
            return False

import random, sys, re

def play(n, m):
    l = [0] * n + [1]
    ls = l * m
    ls += [1] * (n + 1)
    board = Board(ls, n, m)
    turn = FIRST_PLAYER
    while True:
        board.print_board()
        if turn == FIRST_PLAYER:
            print( "FIRST_PLAYER:")
        else:
```

```
            print( "SECOND_PLAYER:")
        if board.gameoverP(turn):
            print( 'Game Over')
            break
        if turn == FIRST_PLAYER:
            while True:
                print( "?=", )
                s = input()
                pat = re.compile('\W+')
                sl = pat.split(s)
                [sn, sm] = sl
                pos = int(sm)*(n+1)+int(sn)
                if board.regalmoveP(turn, pos):
                    board.put_piece(turn, pos)
                    break
        else:
            cand = board.get_candidate(turn)
            ind = random.randint(0, len(cand)-1)
            board.put_piece(turn, cand[ind])
        if turn == FIRST_PLAYER:
            turn = SECOND_PLAYER
        else:
            turn = FIRST_PLAYER

if __name__ == '__main__':
    play(4, 4)
```

です。

今まで作ってきたプログラムに比べると長いですが、ここまで読んできた皆さんにとっては難しくありません。

順番に解説していきます。まずゲームの説明をします。

1. $n \times m$ の長方形の盤を使って、その盤のマス目に碁石を並べていきます。
2. 先手後手交互に、先手は空いているマスに横に連続する 2 マスに碁石を並べ、後手は空いているマスに横に連続する 2 マスに碁石を並べ行きます。
3. 碁石を置けなくなった方が負けです。

注意 ： コンピュータはルールに合致する手をランダムに選んでいるだけですから、碁石を置けなくなった方が勝ちというリバース型のゲームだと思ってもいいです。

上のプログラムを実行すると

```
0 0 0 0
0 0 0 0
0 0 0 0
```

```
0 0 0 0

FIRST_PLAYER:
?=
1 0
0 1 1 0
0 0 0 0
0 0 0 0
0 0 0 0

SECOND_PLAYER:
0 1 1 0
0 0 0 0
0 2 0 0
0 2 0 0

FIRST_PLAYER:
?=
```

のような進行になります。この場合は 4×4 の盤を使っています、座標は $(i, j) i = 0, 1, 2, 3, j = 0, 1, 2, 3$ と 0 から始まっています。後手をコンピュータが持っています。初期画面は全てのマスが空で 0 で埋め尽くされています。?= のプロンプト（先手（FIRST_PLAYER）が手番）で、1 0 と入力しています。これは座標 $(1, 0)$ を指示し、この座標を持つマスとその左のマスに碁石を置くことを指示しています。その直後で碁石を置いた場所が 1 で表示されています。後手（SECOND_PLAYER でコンピュータ）が縦に 2 で表示される位置に碁石を置いています。コンピュータは可能な着手場所の中からランダムに手を選んで置いています。既に共に最善手を尽くせば、後手の勝ちの局面です。私がコンピュータで計算して調べたところでは、5×5 の盤では後手の勝ち、6×6 の盤では先手の勝ちです。

それではプログラムの説明をします。前節の行列のプログラミングと同様に今度はゲームの盤のためのクラスを設計します。

```
class Board:
    def __init__(self, b, n, m):
        self.board = b[:]
        self.width = n
        self.hight = m
```

と定義してあるように、コンストラクターで盤の状態を保持する self.board と盤の横と縦のサイズを保持する self.width, self.hight に値をセットします。関数 play(n, m) の定義

```
def play(n, m):
    l = [0] * n + [1]
    ls = l * m
    ls += [1] * (n + 1)
    board = Board(ls, n, m)
```

でわかるように、このようなゲームではよくやるように、可能な動きかどうか判定しやすいように、盤の大きさを右側と下側に 1 つだけ 1 で埋めて余分に大きくしています。 self.board は単純なリストです。

```
    def print_board(self):
        for y in range(self.hight):
            for x in range(self.width):
                print( self.board[(self.width+1)*y+x],end=' ')
            print()
        print()
```

を Board のメソッドとして追加することにより、局面を表示できるようになりました。着手によって、局面を変化させるメソッドが

```
    def put_piece(self, turn, pos):
        if turn == FIRST_PLAYER:
            self.board[pos] = 1
            self.board[pos+1] = 1
        else:
            self.board[pos] = 2
            self.board[pos+self.width+1] = 2
```

です。put_piece(self, turn, pos) の引数の turn はどちらの手番かを保持しています。
プログラムの先頭で

```
FIRST_PLAYER = 0
SECOND_PLAYER = 1
```

と定数を定義しています。マジックナンバーを使わず、このように定数を使う方がプログラムが読みやすくなります。put_piece(self, turn, pos) の引数 pos は置く 2 個の碁石の左側又は上側の位置のマスのリスト self.board 上のインデクスです。碁石を盤に置けるかどうかを判定する関数が

```
    def regalmoveP(self, turn, pos):
        if turn == FIRST_PLAYER:
            if self.board[pos] == 0 and self.board[pos+1] == 0:
                return True
            else:
                return False
        else:
            if self.board[pos] == 0 and self.board[pos+self.width+1] == 0:
                return True
            else:
                return False
```

です。regalmoveP(self, turn, pos) の引数の意味は put_piece(self, turn, pos) の引数と同じです。self.board の値が 0 かどうかで判定しています。True は真を、 False は偽を表す真理値です。コンピュータの手を決めるために、可能な手を集め、そのインデックスのリストを返す関数が

```
    def get_candidate(self, turn):
        cand = []
        if turn == FIRST_PLAYER:
            for x in range(self.width-1):
                for y in range(self.hight):
                    if (self.regalmoveP(FIRST_PLAYER, (self.width+1)*y+x)):
                        cand.append((self.width+1)*y+x)
        else:
            for x in range(self.width):
                for y in range(self.hight-1):
                    if (self.regalmoveP(SECOND_PLAYER, (self.width+1)*y+x)):
                        cand.append((self.width+1)*y+x)
```

です。もはや、難しい関数ではありません。最後に、ゲームが終了したかどうかを判定する関数が

```
    def gameoverP(self, turn):
        cand = self.get_candidate(turn)
        if len(cand) == 0:
            return True
        else:
            return False
```

です。その局面で可能な手がなければ、ゲームオーバーです。以上で、クラス Board が完成しました。全体は

```
class Board:
    def __init__(self, b, n, m):
        self.board = b[:]
        self.width = n
        self.hight = m
    def put_piece(self, turn, pos):
        if turn == FIRST_PLAYER:
            self.board[pos] = 1
            self.board[pos+1] = 1
        else:
            self.board[pos] = 2
            self.board[pos+self.width+1] = 2
    def regalmoveP(self, turn, pos):
        if turn == FIRST_PLAYER:
            if self.board[pos] == 0 and self.board[pos+1] == 0:
                return True
            else:
                return False
        else:
            if self.board[pos] == 0 and self.board[pos+self.width+1] == 0:
```

```
                return True
            else:
                return False
    def print_board(self):
        for y in range(self.hight):
            for x in range(self.width):
                print( self.board[(self.width+1)*y+x],end=' ')
            print()
        print()
    def get_candidate(self, turn):
        cand = []
        if turn == FIRST_PLAYER:
            for x in range(self.width-1):
                for y in range(self.hight):
                    if (self.regalmoveP(FIRST_PLAYER, (self.width+1)*y+x)):
                        cand.append((self.width+1)*y+x)
        else:
            for x in range(self.width):
                for y in range(self.hight-1):
                    if (self.regalmoveP(SECOND_PLAYER, (self.width+1)*y+x)):
                        cand.append((self.width+1)*y+x)
        return cand
    def gameoverP(self, turn):
        cand = self.get_candidate(turn)
        if len(cand) == 0:
            return True
        else:
            return False
```

です。

次にクラス Board を使って、実際にゲームを進める関数 play(n, m) を定義します。この関数で使う特別な関数を使えるようにするために

```
import random, sys, re
```

のインポートを行います。

関数 play(n, m) の引数 n と m は盤のサイズです。この n と m に基づいて

```
def play(n, m):
    l = [0] * n + [1]
    ls = l * m
    ls += [1] * (n + 1)
    board = Board(ls, n, m)
```

と局面の初期化をまず行います。

```
turn = FIRST_PLAYER
while True:
```

と、手番を先手（ FIRST_PLAYER ）とし、無限ループに入ります。勝負がつけば（ゲームオーバーになれば)、この無限ループを抜け出します。この無限ループの先頭

```
while True:
    board.print_board()
    if turn == FIRST_PLAYER:
        print( "FIRST_PLAYER:")
    else:
        print( "SECOND_PLAYER:")
    if board.gameoverP(turn):
        print( 'Game Over')
        break
```

で、どちらの手番か表示し、ゲームが終わったかどうか調べ、終わっていればそのことを表示し、ループを抜けます。先手番の処理が次に書かれています。

```
    if turn == FIRST_PLAYER:
        while True:
            print( "?=", )
            s = input()
            pat = re.compile('\W+')
            sl = pat.split(s)
            [sn, sm] = sl
            pos = int(sm)*(n+1)+int(sn)
            if board.regalmoveP(turn, pos):
                board.put_piece(turn, pos)
                break
```

ここでは、ルールに合致した手が入力されるまで、無限ループを繰り返す。プロンプト ?= を表示し、キーボードから着点の座標を 'i j" の形式で入力します。

```
            s = input()
            pat = re.compile('\W+')
            sl = pat.split(s)
            [sn, sm] = sl
            pos = int(sm)*(n+1)+int(sn)
```

で、座標を 'i j" の形式で入力してもらったデータから、整数 i と j を取り出すために、正規表現に関するテクニックを使っています。英数字以外が現れたら分割し、英数字だけのリストをつくり、それを取り出し、その座標の self.board 上のインデクス pos を計算しています。Python の正規表現に関する説明はこの本のレベルを超えています。このようにすれば良いとして、真似して使ってください。

```
            if board.regalmoveP(turn, pos):
```

```
                    board.put_piece(turn, pos)
                    break
```

ルールに合致した手であれば、self.board にセットし、内部の無限ルールから抜けます。次に、後手（コンピュータ側）の処理を書いてあります。

```
        else:
            cand = board.get_candidate(turn)
            ind = random.randint(0, len(cand)-1)
            board.put_piece(turn, cand[ind])
```

ルールに合致した手を集め、ランダムに手を選んでいます。コンピュータを強くするためには、この部分を改良すれば良いです。現在は 6×6 の盤なら必勝の手（もしあれば）を探すことが出来ます。最後に、

```
        if turn == FIRST_PLAYER:
            turn = SECOND_PLAYER
        else:
            turn = FIRST_PLAYER
```

で、手番を入れ替えています。

```
if __name__ == '__main__':
    play(4, 4)
```

の呼び出しでゲームを始めます。play(n, m) の引数を変えることで好きな大きさの盤でゲームをすることが出来ます。

3.3 実数値のクラスクラス RealNum

Python には、クラス Decimal が定義されていますが、高速で計算できるアルゴリズムが使われているので

```
>>> print(Decimal(1.2)-Decimal(12.0))
-10.800000000000000044089209850
>>> print(Decimal(1.1)*Decimal(1.1))
1.210000000000000195399252334 03
>>> print(Decimal(12.0)-Decimal(0.120))
11.880000000000000044408920985
>>> print(Decimal(12.0)/Decimal(0.120))
100.000000000000003700743415417
>>>
```

のように、簡単な計算でも誤差が出ます。正確に計算できるようにするには、自分でクラスを設計し、実装すれば良いです。あまりいい実装ではないですがやってみましょう。ここでは、小学校で習い、よく知っている四則のアルゴリズムをプログラミングして、小数点数以下の桁数を固定した

実数の四則演算を備えたクラスを作ってみて、円周率を計算してみましょう。この方法は円周率を1000桁計算するプログラムの模範解答ではありません。専用のプログラムを作ったった方が実行速度は速いはずです。これは馬鹿げた試みだと思うかもしれませんが、模範解答を探し、模範解答をいくら丸暗記しても、決してプログラムは作れるようになりません。よく知っているアルゴリズムを実現するプログラムを、色々な失敗を繰り返しながら、大量に作ってみることにより、プログラムが作れるようになります。

円周率を1000桁計算するプログラムを大きな桁数の固定小数点数のクラスRealNumを作って、計算してみましょう。この節と次節で可能にします。**この節で作るクラスは失敗作です。人の失敗作をわざわざ読みたくないという人は、この節を飛ばして次節に進んでください。人生は失敗の連続です。あらゆる失敗を経験すると最後には人生楽しく生きられます。多分。汎用的な固定小数点数のクラス RealNum を作れば、円周率以外の色々な計算のプログラムが作れるようになります。もっとも、Python のクラス Decimal を使えばいいので、暇でない人は全部飛ばすのが、利口な生き方かもしれません。「車輪を再発見するな」という言葉もあります。これは暇人のためのオブジェクト指向プログラミングの単なる練習です。**

まず、このクラスのコンストラクターと print() 関数を作ります。

```
def set_degit(x, n):
    # 文字列をクラスにセットする数値のリストに変換する
    # 整数部分の桁数と小数点以下の桁数を求める
    if x.count('.') == 0:
        I = len(x)
        m = 0
    else:
        I = x.find('.')
        m =len(x) - I - 1
    # x の数値リストを作る
    val = [0]
    for i, e in enumerate(x):
        if i < I:
            val.append(int(e))
        elif i == I:
            continue
        elif i <= I + n:
            val.append(int(e))
        else:
            break
    # 小数点以下の桁数の補正
    if n > m:
        for i in range(n-m):
            val.append(0)
    return val
def get_complement(val):
    # 10 の補数に変換する
```

```
    # val は正数か負数か
    if val[0] == 0:
        flag = True
    else:
        flag = False
    res = []
    # 右端の 0 の個数を計算する
    z = 0
    for i in range(len(val)):
        if val[len(val)-i-1] == 0:
            z += 1
        else:
            break
    # 10 の補数を取る
    for i, e in enumerate(val):
        if i == 0:
            if flag == True:
                res.append(9)
            else:
                res.append(0)
        elif i < len(val) - z - 1:
            res.append(9 - e)
        elif i == len(val) - z -1:
            res.append(10 - e)
        else:
            res.append(0)
    return res
class RealNum:
    def __init__(self, x, n):
        x = x.strip()
        p = x.count('.')
        q = x.count('0')
        if x[0] == '-': # 負数
            x = x.lstrip('-')
            # 文字列をクラスにセットする数値のリストに変換する
            val = set_degit(x, n)
            self.val = get_complement(val) # 10 の補数に変換する
        elif p+q == len(x): # ゼロ
            self.val = [0, 0]
            for i in range(n):
                self.val.append(0)
        else: # 正数
            # 文字列をクラスにセットする数値のリストに変換する
```

```
            self.val = set_degit(x, n)
        self.n = n
    def print(self):
        if self.val[0] == 0: # 非負数
            val = self.val[:]
        else: # 負数
            val = get_complement(self.val)
        m = len(val)
        for i, e in enumerate(val):
            if i == 0:
                if self.val[0] == 0:
                    print(' ', end=' ')
                else:
                    print('-', end=' ')
            elif i == m-self.n:
                print('.', e, end=' ')
            else:
                print(e, end=' ')
        print()
if __name__ == '__main__':
    X = RealNum('12.53', 4)
    X.print()
    Y = RealNum('-53.47', 4)
    Y.print()
    Z = RealNum('0', 4)
    Z.print()
```

実行すると

```
  1 2 . 5 3 0 0
- 5 3 . 4 7 0 0
  0 . 0 0 0 0
```

です。上手くいっているようです。

コンストラクターでは、文字列の数値と小数点以下の桁数を与えます。クラス RealNum では、10 進数の数値のリスト self.val と小数点以下の桁数 self.n を保持します。例えば 12.3000 の場合は、self.val には実数を構成する数字の列 self.val = [0, 1, 2, 3, 0, 0, 0])、self.n には小数点以下の桁数 self.n=4 をセットします。負の数と区別するために、正の数にはリストの先頭に 0 を追加します。0.012.3 の場合は、self.val には実数を構成する数字の列 self.val = [0, 0, 1, 2, 3])、self.n には小数点以下の桁数 self.n=4 をセットします。（このようにしたのは「**C -言語とプログラミング-**」米田信夫編　産業図書の円周率の計算のプログラムのイメージが残っていたからです。別のやり方：「**0.012.3** の場合は、**self.val** には実数を構成する数字の列 **self.val = [0, 1, 2, 3]**)、**0.0012.** の場合は、**self.val** には実数を構成する数字の列 **self.val = [0, 1, 2,]**)、**self.n** には小数点以下の桁数 **self.n=4** をセットします。」もあります。）

負の数は 1 0 の補数で表します。例えば、-12.3 は

99.99-12.3+0.1 = 87.7
と計算し、負の数でわかるように、頭に 9 を追加し、987.7 とし、n=4 なら、self.val = [9,8,7,7,0,0,0], self.n = 4 とします。987.65 の１０の補数をとると、
999.99-987.66+0.01=012.34
と元に戻ります。
言い方を変えれば、負の数 -12.3000 は 10 の補数を使って表現します。つまり、100.0000 を加えます。

$$\begin{aligned} -12.3000 + 100.0000 &= -12.3000 + 99.9999 + 0.0001 \\ &= (99.9999 - 12.3000) + 0.0001 \\ &= 87.6999 + 0.0001 \\ &= 87.7000 \end{aligned}$$

そして、[8, 7, 7, 0, 0, 0] の先頭に 9 を追加して、[9, 8, 7, 7, 0, 0, 0] で負の数であることを表すことにします。元に戻すには、100.0000 - 87.7000 を計算すれば良いです。

$$\begin{aligned} 100.0000 - 87.7000 &= 99.9999 + 0.0001 - 87.7000 \\ &= (99.9999 - 87.7000) + 0.0001 \\ &= 12.2999 + 0.0001 \\ &= 12.3000 \end{aligned}$$

です。二つを合わせると

$$\begin{aligned} -12.3000 &= -12.3000 + 100.0000 - 100.0000 \\ &= -12.3000 + 99.9999 + 0.0001 - 100.0000 \\ &= (99.9999 - 12.3000) + 0.0001 - 100.0000 \\ &= 87.6999 + 0.0001 - 100.0000 \\ &= 87.7000 - 100.0000 \\ &= -(100.0000 - 87.7000 \\ &= -(99.9999 + 0.0001 - 87.7000) \\ &= -((99.9999 - 87.7000) + 0.000)1 \\ &= -(12.2999 + 0.0001) \\ &= -12.3000 \end{aligned}$$

です。

複雑なように見えますが、このようにすると足し算で、正の数と負の数との場合分けが必要でなく処理が簡単になります。

$$\begin{aligned} 123.4500 + (-12.3000) &= 123.4500 + (-12.3000 + 1000.0000) - 1000.0000 \\ &= (123.4500 + 987.7000) - 1000.0000 \\ &= 1111.1500 - 1000.0000 \\ &= 1111.1500 \end{aligned}$$

の計算から分かるように、負の数の計算も１０の補数で表現しておけば、そのまま足し算すれば良いです。オーバーフローを適切に処理すれば良いです。

計算機の内部でも２の補数を使って表現しています。全体の桁数（整数部分の桁数）も制限すると単純に表現でき、円周率の計算のためにはそれで充分ですが、整数部分はいくらでも拡大可能の方が利用がって良いだろうと思ったのでこのような設計にしました。この pdf は最善のアルゴリズムを紹介するのではなく、自分で思いついたアルゴリズムをどのようにプログラミングしたらよいか解説するのが目的です。良い思い付きでなければ、以下でやっているように苦労することになりますが、多分、ともかく正しく計算してくれるプログラムをでっちあげてみることが勉強にはなります。

ゼロは正の数の仲間として、self.val = [0, 0, 0, 0 0, 0], self.n = 4 と表す方法と特別なものとして、self.val = [0, 0, 0 0, 0], self.n = 4 と表す方法が考えられますが、ここでは前者の方法を採用します。どっちが良い選択かは、両方のプログラムを作ってみればわかります。練習のためには、後者を採用すればどのようにプログラミングすれば良いかやってみると良いです。

コンストラクターは

```
def __init__(self, x, n):
    x = x.strip()
    p = x.count('.')
    q = x.count('0')
    if x[0] == '-': # 負数
        x = x.lstrip('-')
        # 文字列をクラスにセットする数値のリストに変換する
        val = set_degit(x, n)
        self.val = get_complement(val) # 10 の補数に変換する
    elif p+q == len(x): # ゼロ
        self.val = [0, 0]
        for i in range(n):
            self.val.append(0)
    else: # 正数
        # 文字列をクラスにセットする数値のリストに変換する
        self.val = set_degit(x, n)
    self.n = n
```

と定義しています。

```
    x = x.strip()
```

```
        p = x.count('.')
        q = x.count('0')
```

で、前後の空白を除き、小数点の個数（小数点があるかどうかを知るため)、ゼロの個数（ゼロかどうか調べるため）をまず調べています。残りの部分

```
        if x[0] == '-': # 負数
            x = x.lstrip('-')
            # 文字列をクラスにセットする数値のリストに変換する
            val = set_degit(x, n)
            self.val = get_complement(val) # 10 の補数に変換する
        elif p+q == len(x): # ゼロ
            self.val = [0, 0]
            for i in range(n):
                self.val.append(0)
        else: # 正数
            # 文字列をクラスにセットする数値のリストに変換する
            self.val = set_degit(x, n)
        self.n = n
```

で、負の数かゼロか正数かで場合分けし、セットすべき数の列を作り、self.val にセットし、最後に小数点以下の桁数 n を self.n にセットしています。クラス RealNum のメンバーは self.val と self.n だけです。

ゼロの場合は、単に決めた通りの数字の列をセットするだけです。

正の数の場合は、関数 `set_degit(x, n)`

```
def set_degit(x, n):
    # 文字列をクラスにセットする数値のリストに変換する
    # 整数部分の桁数と小数点以下の桁数を求める
    if x.count('.') == 0:
        I = len(x)
        m = 0
    else:
        I = x.find('.')
        m =len(x) - I - 1
    # x の数値リストを作る
    val = [0]
    for i, e in enumerate(x):
        if i < I:
            val.append(int(e))
        elif i == I:
            continue
        elif i <= I + n:
            val.append(int(e))
        else:
```

```
            break
    # 小数点以下の桁数の補正
    if n > m:
        for i in range(n-m):
            val.append(0)
    return val
```

を使ってセットすべき数の列を作ります。変数 I には整数部分の桁数を、m には小数点以下の桁数を求めるセットしています。正の数を表す 0 を先頭に置き、与えられた文字列から数字のリストを作っています。与えられた小数点以下の桁数 n より与えられた文字列の小数点以下の桁数が大きい場合にはその部分は切り捨てています。

正の数の場合は、絶対値を取り出し、関数 `set_degit(x, n)` で数値のリストを作り、関数 `get_complement(val)`

```
def get_complement(val):
    # 10 の補数に変換する
    # val は正数か負数か
    if val[0] == 0:
        flag = True
    else:
        flag = False
    res = []
    # 右端の 0 の個数を計算する
    z = 0
    for i in range(len(val)):
        if val[len(val)-i-1] == 0:
            z += 1
        else:
            break
    # 10 の補数を取る
    for i, e in enumerate(val):
        if i == 0:
            if flag == True:
                res.append(9)
            else:
                res.append(0)
        elif i < len(val) - z - 1:
            res.append(9 - e)
        elif i == len(val) - z -1:
            res.append(10 - e)
        else:
            res.append(0)
    return res
```

で 10 の補数に変換します。後で負の数を求めるときや引き算で使うために、正の数だけでなく負の数の変換もできるようにしています。ゼロの時、変換は不要であるということをプログラミングしておいた方が一般的かも分かりません。そうしておけば、負の数を求めるとき、場合分けが不要になります。

次に足し算を出来るようにします。必要なプログラミングは

```
def convert_pos(c, n):
    # 正数のリストを文字列に変換する
    c.insert(n, '.')
    c.reverse()
    C = ''.join(c)
    return C
def convert_neg(c, n):
    # 負数のリストを文字列に変換する
    c.reverse()
    D = []
    for e in c:
        D.append(int(e))
    C = get_complement(D)
    if len(C) > n+1:
        del C[0]
    else:
        C[0] = 0
    r = ['-']
    for i, e in enumerate(C):
        if i == len(C)-n:
            r.append('.')
        r.append(str(e))
    r = ''.join(r)
    return r
def add(a, b):
    A = a.val[:]
    B = b.val[:]
    c = []
    A.reverse()
    B.reverse()
    # 整数部分の桁数を揃える
    if len(A) > len(B):
        if B[len(B)-1] == 0:
            for i in range(len(A)-len(B)):
                B.append(0)
        elif B[len(B)-1] == 9:
            for i in range(len(A)-len(B)):
```

```
            B.append(9)
elif len(A) < len(B):
    if A[len(A)-1] == 0:
        for i in range(len(B)-len(A)):
            A.append(0)
    elif A[len(A)-1] == 9:
        for i in range(len(B)-len(A)):
            A.append(9)
# 和の計算
t = 0
for i in range(len(A)):
    x = A[i] + B[i] + t
    c.append(str(x % 10))
    t = x // 10
# 計算結果の整理
if c[len(c)-1] == '1':
    # 正数、繰り上がり
    C = convert_pos(c, a.n) # 正数のリストを文字列に変換する
elif c[len(c)-1] == '0':
    # 正数又は0、余分な0を除く
    p = len(c)
    del c[p-1]
    for i in range(len(c)-1, a.n, -1):
        if c[i] == '0':
            p -= 1
        else:
            break
    del c[p-1:]
    C = convert_pos(c, a.n) # 正数のリストを文字列に変換する
elif c[len(c)-1] == '8':
    # 負数、繰り上がり
    c.append('9')
    C = convert_neg(c, a.n) # 負数のリストを文字列に変換する
elif c[len(c)-1] == '9':
    # 負数、余分な9を除く
    p = len(c)
    for i in range(len(c)-1, a.n, -1):
        if c[i] == '9' and c[i-1] == '9':
            p -= 1
        else:
            break
    del c[p:]
    C = convert_neg(c, a.n) # 負数のリストを文字列に変換する
```

```
    # オブジェクトを作成
    return RealNum(C, a.n)
```

です。a + b の表現を可能にするために、演算子のオーバーロードをします。クラス RealNum の定義に

```
    __add__ = add
```

を追加します。

```
if __name__ == '__main__':
    X = RealNum('5.2', 4); X.print()
    Y = RealNum('4.5', 4); Y.print()
    Z = add(X, Y)
    print('RESULT = ', end=" "); Z.print()
    X = RealNum('5.2', 4); X.print()
    Y = RealNum('6.9', 4); Y.print()
    Z = add(X, Y)
    print('RESULT = ', end=" "); Z.print()
    X = RealNum('5.5', 2); X.print()
    Y = RealNum('-7.8', 2); Y.print()
    Z = add(X, Y)
    print('RESULT = ', end=" "); Z.print()
    X = RealNum('8.6', 2); X.print()
    Y = RealNum('-7.5', 2); Y.print()
    Z = X + Y
    print('RESULT = ', end=" "); Z.print()
    X = RealNum('-7.7', 2); X.print()
    Y = RealNum('-0.1', 2); Y.print()
    Z = X + Y
    print('RESULT = ', end=" "); Z.print()
    X = RealNum('-7.6', 2); X.print()
    Y = RealNum('-5.8', 2); Y.print()
    Z = X + Y
    print('RESULT = ', end=" "); Z.print()
```

を実行すると

```
  5 . 2 0 0 0
  4 . 5 0 0 0
RESULT =    9 . 7 0 0 0
  5 . 2 0 0 0
  6 . 9 0 0 0
RESULT =    1 2 . 1 0 0 0
  5 . 5 0
- 7 . 8 0
```

```
RESULT =  - 2 . 3 0
  8 . 6 0
- 7 . 5 0
RESULT =    1 . 1 0
- 7 . 7 0
- 0 . 1 0
RESULT =  - 7 . 8 0
- 7 . 6 0
- 5 . 8 0
RESULT =  - 1 3 . 4 0
```

となります。

関数 add(a, b) は

```
def add(a, b):
    A = a.val[:]
    B = b.val[:]
    c = []
    A.reverse()
    B.reverse()
    # 整数部分の桁数を揃える
    if len(A) > len(B):
        if B[len(B)-1] == 0:
            for i in range(len(A)-len(B)):
                B.append(0)
        elif B[len(B)-1] == 9:
            for i in range(len(A)-len(B)):
                B.append(9)
    elif len(A) < len(B):
        if A[len(A)-1] == 0:
            for i in range(len(B)-len(A)):
                A.append(0)
        elif A[len(A)-1] == 9:
            for i in range(len(B)-len(A)):
                A.append(9)
    # 和の計算
    t = 0
    for i in range(len(A)):
        x = A[i] + B[i] + t
        c.append(str(x % 10))
        t = x // 10
    # 計算結果の整理
    if c[len(c)-1] == '1':
        # 正数、繰り上がり
```

```
        C = convert_pos(c, a.n) # 正数のリストを文字列に変換する
    elif c[len(c)-1] == '0':
        # 正数又は0、余分な0を除く
        p = len(c)
        del c[p-1]
        for i in range(len(c)-1, a.n, -1):
            if c[i] == '0':
                p -= 1
            else:
                break
        del c[p-1:]
        C = convert_pos(c, a.n) # 正数のリストを文字列に変換する
    elif c[len(c)-1] == '8':
        # 負数、繰り上がり
        c.append('9')
        C = convert_neg(c, a.n) # 負数のリストを文字列に変換する
    elif c[len(c)-1] == '9':
        # 負数、余分な9を除く
        p = len(c)
        for i in range(len(c)-1, a.n, -1):
            if c[i] == '9' and c[i-1] == '9':
                p -= 1
            else:
                break
        del c[p:]
        C = convert_neg(c, a.n) # 負数のリストを文字列に変換する
    # オブジェクトを作成
    return RealNum(C, a.n)
```

で、かなり複雑そうに見えますが、足し算の為のアルゴリズムは

```
    # 和の計算
    t = 0
    for i in range(len(A)):
        x = A[i] + B[i] + t
        c.append(str(x % 10))
        t = x // 10
```

の部分で、これは小学校で習う足し算のアルゴリズムをプログラミングしているだけです。 a, b の整数部分の桁数を揃え、和を計算すると、オーバーフローの値 t と計算結果の最上位の数値 c[len(c)-1] の組み合わせは

1. t = 0, c[len(c)-1] =0
2. t = 0, c[len(c)-1] =1

3. t = 0, c[len(c)-1] =9
4. t = 1, c[len(c)-1] =0
5. t = 1, c[len(c)-1] =9
6. t = 1, c[len(c)-1] =8

の場合があります。上の実行結果はこの場合分けに対応しています。この場合分けに応じて、計算結果を整理して、文字列に変換して、RealNum のオブジェクトを作れば良いです。関数 add(a, b) のアルゴリズムが正しいことは、例えば、正の数と負の数の和は

$$\begin{aligned} 360 - 123 &= 360 - 123 + 1000 - 1000 \\ &= 360 - 123 + 999 + 1 - 1000 \\ &= 360 + (999 - 123 + 1) - 1000 \\ &= (360 + 877) - 1000 \\ &= 1237 - 1000 \\ &= 237 \end{aligned}$$

の計算などで理解できると思います。それぞれの場合を、上手く整理して、コツコツプログラミングしていけばいいです。

```
def convert_pos(c, n):
    # 正数のリストを文字列に変換する
    c.insert(n, '.')
    c.reverse()
    C = ''.join(c)
    return C
def convert_neg(c, n):
    # 負数のリストを文字列に変換する
    c.reverse()
    D = []
    for e in c:
        D.append(int(e))
    C = get_complement(D)
    if len(C) > n+1:
        del C[0]
    else:
        C[0] = 0
    r = ['-']
    for i, e in enumerate(C):
        if i == len(C)-n:
            r.append('.')
        r.append(str(e))
```

```
    r = ''.join(r)
    return r
```

は、それぞれ、正数のリストを文字列に変換する関数と負数のリストを文字列に変換する関数です。
例えば、この和を計算する部分を作っていてうまく動かないときは、

```
    # 和の計算
    t = 0
    for i in range(len(A)):
        x = A[i] + B[i] + t
        c.append(str(x % 10))
        t = x // 10
    # 計算結果の整理
```

の部分に

```
    # 和の計算
    t = 0
    for i in range(len(A)):
        x = A[i] + B[i] + t
        c.append(str(x % 10))
        t = x // 10
    # 計算結果の整理
    print('t=', t)
    print('c=', c)
```

のように print() 関数を入れ、何が起こっているか調べ、どのように計算結果を整理したらよいか考えれば良いです。ここでは、上で述べた 6 通りの場合が起きる計算をしてみて調整します。
数字のリストで計算し、**RealNum** のオブジェクトを作るためには、一々文字列に変換する必要があります。数字のリストを渡して、**RealNum** のオブジェクトを作った方が効率が良いです。このまま続けるのが馬鹿らしくなった方は、次節に飛んでもかまいませんが、プログラミングの勉強にはならないと思います。自分でプログラミングが出来るようになりたいと思ったら、やりだしたら最後までやってみることだと思います。得るところは大いにあると思います。私はこのようなプログラムをいっぱい作ってみることにより、自由にプログラミングできるようになりました。あらゆる失敗を経験して、プログラミングが出来るようになります。しかし、効率の良い **realNum** の模範解答だけを知りたい人は、**Python** の特殊性を使った最後の節に飛んでください。
次に差 a - b を計算できるようにしましょう。

```
def zeroP(a):
    A = a.val[:]
    for e in A:
        if e != 0: return False
    return True
def neg(a):
    A = a.val[:]
    if A[0] == 0:
```

```
        if zeroP(a):
            return a
        del A[0]
        N = []
        N.append('-')
        for i, e in enumerate(A):
            if i == len(A)-a.n:
                N.append('.')
            N.append(str(e))
        M = ''.join(N)
    else:
        B = get_complement(A)
        del B[0]
        N = []
        for i, e in enumerate(B):
            if i == len(B)-a.n:
                N.append('.')
            N.append(str(e))
        M = ''.join(N)
    return RealNum(M, a.n)
```

と定義し、

```
    __neg__ = neg
    def __sub__(a, b):
        return a + (-b)
```

をクラス RealNum に追加します。

```
if __name__ == '__main__':
    X = RealNum('-7.6', 2); X.print()
    Y = RealNum('5.8', 2); Y.print()
    A = -X; A.print()
    B = -Y; B.print()
    Z = X - Y
    print('RESULT = ', end=" "); Z.print()
```

を実行すると

```
- 7 . 6 0
  5 . 8 0
  7 . 6 0
- 5 . 8 0
RESULT =  - 1 3 . 4 0
```

となります。

関数 neg(a)

```
def neg(a):
    A = a.val[:]
    if A[0] == 0:
        if zeroP(a):
            return a
        del A[0]
        N = []
        N.append('-')
        for i, e in enumerate(A):
            if i == len(A)-a.n:
                N.append('.')
            N.append(str(e))
        M = ''.join(N)
    else:
        B = get_complement(A)
        del B[0]
        N = []
        for i, e in enumerate(B):
            if i == len(B)-a.n:
                N.append('.')
            N.append(str(e))
        M = ''.join(N)
    return RealNum(M, a.n)
```

は、ゼロまたは正の数の時は簡単で、そのまま返すか、負の符号を付けて文字列に変換し、コンストラクターに渡しオブジェクトを作ればいいです。負の数の時は、`get_complement(A)` で 10 の補数に変換した後、文字列に変換し、コンストラクターに渡しオブジェクトを作っています。単項演算子の - ができれば、引き算は

```
    def __sub__(a, b):
        return a + (-b)
```

でいいです。

次に積を定義します。

```
def sub_mul(a, b):
    # a > 0 and b > 0 と仮定する
    # c = a * b を計算する
    A = a.val[:]
    B = b.val[:]
    del A[0]
    del B[0]
    A.reverse()
    B.reverse()
    N = len(A) + len(B)
```

```
    C = [0] * N
    for i, e in enumerate(B):
        t = 0
        for j, f in enumerate(A):
            g = C[i+j] + e * f + t
            t = g // 10
            C[i+j] = g % 10
        C[i+len(A)] = t
    if C[len(C)-1] == 0:
        del C[len(C)-1]
    C.reverse()
    M = []
    for i, e in enumerate(C):
        if e == 0 and i < len(C)-2*a.n-1:
            continue
        elif i == len(C)-2*a.n:
            M.append('.')
            M.append(str(e))
        elif i > len(C)-a.n:
            break
        else:
            M.append(str(e))
    L = ''.join(M)
    return RealNum(L, a.n)
def mul(a, b):
    if a.val[0] == 0 and b.val[0] == 0: # 正数×正数
        c = sub_mul(a, b)
    elif a.val[0] == 0 and b.val[0] == 9: # 正数×負数
        d = -b
        e = sub_mul(a, d)
        c = -e
    elif a.val[0] == 9 and b.val[0] == 0: # 負数×正数
        d = -a
        e = sub_mul(d, b)
        c = -e
    elif a.val[0] == 9 and b.val[0] == 9: # 負数×負数
        d = -a
        e = -b
        c = sub_mul(d, e)
    return c
```

と定義し、クラス RealNum の定義に

```
    __mul__ = mul
```

追加し、

```
    X = RealNum('7.3', 2); X.print()
    Y = RealNum('5.2', 2); Y.print()
    Z = mul(X, Y)
    print('RESULT = ', end=" "); Z.print()
    X = RealNum('-7.3', 2); X.print()
    Y = RealNum('5.2', 2); Y.print()
    Z = mul(X, Y)
    print('RESULT = ', end=" "); Z.print()
    X = RealNum('7.3', 2); X.print()
    Y = RealNum('-5.2', 2); Y.print()
    Z = X * Y
    print('RESULT = ', end=" "); Z.print()
    X = RealNum('-7.3', 2); X.print()
    Y = RealNum('-5.2', 2); Y.print()
    Z = X * Y
    print('RESULT = ', end=" "); Z.print()
```

を実行すると

```
  7 . 3 0
  5 . 2 0
RESULT =    3 7 . 9 6
- 7 . 3 0
  5 . 2 0
RESULT =  - 3 7 . 9 6
  7 . 3 0
- 5 . 2 0
RESULT =  - 3 7 . 9 6
- 7 . 3 0
- 5 . 2 0
RESULT =    3 7 . 9 6
```

となります。

関数 sub_mul(a, b) の主要部分

```
def sub_mul(a, b):
    # a > 0 and b > 0 と仮定する
    # c = a * b を計算する
    A = a.val[:]
    B = b.val[:]
    del A[0]
    del B[0]
    A.reverse()
```

```
    B.reverse()
    N = len(A) + len(B)
    C = [0] * N
    for i, e in enumerate(B):
        t = 0
        for j, f in enumerate(A):
            g = C[i+j] + e * f + t
            t = g // 10
            C[i+j] = g % 10
        C[i+len(A)] = t
    if C[len(C)-1] == 0:
        del C[len(C)-1]
    C.reverse()
```

のアルゴリズムは、小学校で学ぶ積のアルゴリズムです。このようにして積を計算した後は、関数 sub_mul(a, b) の後半

```
    M = []
    for i, e in enumerate(C):
        if e == 0 and i < len(C)-2*a.n-1:
            continue
        elif i == len(C)-2*a.n:
            M.append('.')
            M.append(str(e))
        elif i > len(C)-a.n:
            break
        else:
            M.append(str(e))
    L = ''.join(M)
    return RealNum(L, a.n)
```

で、小数点以下の桁数が n になるよう切り捨てて、文字列に変換して、コンストラクターにわたし、オブジェクトをつくっています。

関数 mul(a, b)

```
def mul(a, b):
    if a.val[0] == 0 and b.val[0] == 0: # 正数×正数
        c = sub_mul(a, b)
    elif a.val[0] == 0 and b.val[0] == 9: # 正数×負数
        d = -b
        e = sub_mul(a, d)
        c = -e
    elif a.val[0] == 9 and b.val[0] == 0: # 負数×正数
        d = -a
        e = sub_mul(d, b)
```

```
        c = -e
    elif a.val[0] == 9 and b.val[0] == 9: # 負数×負数
        d = -a
        e = -b
        c = sub_mul(d, e)
    return c
```

は、場合分けして、関数 sub_mul(a, b) で計算しているだけです。

割り算の単純なアルゴリズムを実現するために、比較演算子 == と <= と >= と < と > をオーバーロードしておきましょう。

```
def eq(a, b):
    c = a - b
    return zeroP(c)
def gt(a, b):
    c = b - a
    if c.val[0] == 9:
        return True
    else:
        return False
def lt(a, b):
    c = a - b
    if c.val[0] == 9:
        return True
    else:
        return False
```

を定義し、クラス RealNum に

```
    def __le__(a, b):
        c = b-a
        if c.val[0] == 0:
            return True
        else:
            return False
    def __ge__(self, other):
        c = self - other
        if c.val[0] == 0:
            return True
        else:
            return False
    __gt__ = gt
    __lt__ = lt
    __eq__ = eq
```

を追加します。どちらの形で定義してもいいみたいです。

```
if __name__ == '__main__':
    X = RealNum('7.3', 2); X.print()
    Y = RealNum('5.2', 2); Y.print()
    print('RESULT = ', X <= Y)
    print('RESULT = ', X >= Y)

    X = RealNum('-7.3', 2); X.print()
    Y = RealNum('5.2', 2); Y.print()
    print('RESULT = ', X <= Y)
    print('RESULT = ', X >= Y)
```

を実行します。

```
  7 . 3 0
  5 . 2 0
RESULT =  False
RESULT =  True
- 7 . 3 0
  5 . 2 0
RESULT =  True
RESULT =  False
```

となります。
まず、補助関数

```
def single(x, m, n):
    if m > n:
        s = [str(x)]
        for i in range(m-1):
            if i == m-n-1:
                s.append('.')
            s.append('0')
        return RealNum(''.join(s), n)
    elif m <= n:
        s = ['0','.']
        for i in range(n):
            if i == n - m:
                s.append(str(x))
            else:
                s.append('0')
        r = RealNum(''.join(s), n)
        return r
```

を定義します。

```
if __name__ == '__main__':
```

```
    X = single_num(4, 4, 2)
    X.print()
```

を実行します。

```
  4 0 . 0 0
```

となりす。補助関数 single_num(x, m, n) で、 m > n の場合は、 m は全体の桁数、n は小数点以下の桁数、x は最初の数字で、m <= n の場合は、n は小数点以下の桁数、 少数第 m 位が x である RealNum のオブジェクトを返します。

以上の準備のもと、割り算を定義します。割り算の効率の良いアルゴリズムは Knuth 著「The Art of Computer Programming」 にアルゴリズムが載っていますが、ここでは小学生がやるような試行錯誤による素朴なアルゴリズムで実現します。

次のように定義します

```
def copy(a):
    A = a.val[:]
    del A[0]
    c = []
    for i, e in enumerate(A):
        if i == len(A)-a.n:
            c.append('.')
        c.append(str(e))
    return RealNum(''.join(c), a.n)
    def sub_div(a, b):
    # a > 0 and b > 0 と仮定する
    # q に a / b の商の数値をセット
    q = []
    m = len(a.val)-1
    d = copy(a)
    for i in range(m):
        c = 1
        p = single(c, m-i, a.n)
        if b * p > d:
            q.append(0)
            continue
        while b * p <= d:
            c += 1
            p = single(c, m-i, a.n)
        c -= 1
        p = single(c, m-i, a.n)
        q.append(c)
        d = d - b * p
    # 不要な 0 を除く
    z = 0
```

```
    for i in range(m):
        if q[i] != 0:
            z = i
            break
        elif i < m - a.n -1: continue
        elif i == m - a.n - 1:
            z = i
            break
    # 結果を文字列に変換
    Q = []
    for i, e in enumerate(q):
        if i < z: continue
        if i == m - a.n:
            Q.append('.')
        Q.append(str(e))
    return RealNum(''.join(Q), a.n)
def div(a, b):
    if a.val[0] == 0 and b.val[0] == 0: # 正数/正数
        c = sub_div(a, b)
    elif a.val[0] == 0 and b.val[0] == 9: # 正数/負数
        d = -b
        e = sub_div(a, d)
        c = -e
    elif a.val[0] == 9 and b.val[0] == 0: # 負数/正数
        d = -a
        e = sub_div(d, b)
        c = -e
    elif a.val[0] == 9 and b.val[0] == 9: # 負数/負数
        d = -a
        e = -b
        c = sub_div(d, e)
    return c
```

そして、クラス RealNum に

```
    __truediv__ = div
```

を追加します。

```
if __name__ == '__main__':
    X = RealNum('7.3', 2); X.print()
    Y = RealNum('5', 2); Y.print()
    Z = X / Y
    print('RESULT = ', end=" "); Z.print()
    X = RealNum('-7.3', 2); X.print()
```

```
    Y = RealNum('5', 2); Y.print()
    Z = X / Y
    print('RESULT = ', end=" "); Z.print()
    X = RealNum('7.3', 2); X.print()
    Y = RealNum('-5', 2); Y.print()
    Z = X / Y
    print('RESULT = ', end=" "); Z.print()
    X = RealNum('-7.3', 2); X.print()
    Y = RealNum('-5', 2); Y.print()
    Z = X / Y
    print('RESULT = ', end=" "); Z.print()
```

を実行します。

```
  7 . 3 0
  5 . 0 0
RESULT =    1 . 4 6
- 7 . 3 0
  5 . 0 0
RESULT =  - 1 . 4 6
  7 . 3 0
- 5 . 0 0
RESULT =  - 1 . 4 6
- 7 . 3 0
- 5 . 0 0
RESULT =    1 . 4 6
```

となります。

関数 sub_div(a, b)

```
def sub_div(a, b):
    # a > 0 and b > 0 と仮定する
    # q に a / b の商の数値をセット
    q = []
    m = len(a.val)-1
    d = copy(a)
    for i in range(m):
        c = 1
        p = single(c, m-i, a.n)
        if b * p > d:
            q.append(0)
            continue
        while b * p <= d:
            c += 1
            p = single(c, m-i, a.n)
```

```
            c -= 1
            p = single(c, m-i, a.n)
            q.append(c)
            d = d - b * p
        # 不要な 0 を除く
        z = 0
        for i in range(m):
            if q[i] != 0:
                z = i
                break
            elif i < m - a.n -1: continue
            elif i == m - a.n - 1:
                z = i
                break
        # 結果を文字列に変換
        Q = []
        for i, e in enumerate(q):
            if i < z: continue
            if i == m - a.n:
                Q.append('.')
            Q.append(str(e))
        return RealNum(''.join(Q), a.n)
```

が小学校で習う割り算のアルゴリズムをそのままプログラミングしたものです。

これでクラス RealNum の定義が終わりました。

```
def set_degit(x, n):
    # 文字列をクラスにセットする数値のリストに変換する
    # 整数部分の桁数と小数点以下の桁数を求める
    if x.count('.') == 0:
        I = len(x)
        m = 0
    else:
        I = x.find('.')
        m =len(x) - I - 1
    # x の数値リストを作る
    val = [0]
    for i, e in enumerate(x):
        if i < I:
            val.append(int(e))
        elif i == I:
            continue
        elif i <= I + n:
            val.append(int(e))
```

```
            else:
                break
        # 小数点以下の桁数の補正
        if n > m:
            for i in range(n-m):
                val.append(0)
        return val

    def get_complement(val):
        # 10 の補数に変換する
        # val は正数か負数か
        if val[0] == 0:
            flag = True
        else:
            flag = False
        res = []
        # 右端の 0 の個数を計算する
        z = 0
        for i in range(len(val)):
            if val[len(val)-i-1] == 0:
                z += 1
            else:
                break
        # 10 の補数を取る
        for i, e in enumerate(val):
            if i == 0:
                if flag == True:
                    res.append(9)
                else:
                    res.append(0)
            elif i < len(val) - z - 1:
                res.append(9 - e)
            elif i == len(val) - z -1:
                res.append(10 - e)
            else:
                res.append(0)
        return res
    def convert_pos(c, n):
        # 正数のリストを文字列に変換する
        c.insert(n, '.')
        c.reverse()
        C = ''.join(c)
        return C
```

```
def convert_neg(c, n):
    # 負数のリストを文字列に変換する
    c.reverse()
    D = []
    for e in c:
        D.append(int(e))
    C = get_complement(D)
    if len(C) > n+1:
        del C[0]
    else:
        C[0] = 0
    r = ['-']
    for i, e in enumerate(C):
        if i == len(C)-n:
            r.append('.')
        r.append(str(e))
    r = ''.join(r)
    return r

def add(a, b):
    A = a.val[:]
    B = b.val[:]
    c = []
    A.reverse()
    B.reverse()
    # 整数部分の桁数を揃える
    if len(A) > len(B):
        if B[len(B)-1] == 0:
            for i in range(len(A)-len(B)):
                B.append(0)
        elif B[len(B)-1] == 9:
            for i in range(len(A)-len(B)):
                B.append(9)
    elif len(A) < len(B):
        if A[len(A)-1] == 0:
            for i in range(len(B)-len(A)):
                A.append(0)
        elif A[len(A)-1] == 9:
            for i in range(len(B)-len(A)):
                A.append(9)
    # 和の計算
    t = 0
    for i in range(len(A)):
```

```
            x = A[i] + B[i] + t
            c.append(str(x % 10))
            t = x // 10
        # 計算結果の整理
        if c[len(c)-1] == '1':
            # 正数、繰り上がり
            C = convert_pos(c, a.n) # 正数のリストを文字列に変換する
        elif c[len(c)-1] == '0':
            # 正数又は0、余分な0を除く
            p = len(c)
            del c[p-1]
            for i in range(len(c)-1, a.n, -1):
                if c[i] == '0':
                    p -= 1
                else:
                    break
            del c[p-1:]
            C = convert_pos(c, a.n) # 正数のリストを文字列に変換する
        elif c[len(c)-1] == '8':
            # 負数、繰り上がり
            c.append('9')
            C = convert_neg(c, a.n) # 負数のリストを文字列に変換する
        elif c[len(c)-1] == '9':
            # 負数、余分な9を除く
            p = len(c)
            for i in range(len(c)-1, a.n, -1):
                if c[i] == '9' and c[i-1] == '9':
                    p -= 1
                else:
                    break
            del c[p:]
            C = convert_neg(c, a.n) # 負数のリストを文字列に変換する
        # オブジェクトを作成
        return RealNum(C, a.n)

    def neg(a):
        A = a.val[:]
        if A[0] == 0:
            del A[0]
            N = []
            N.append('-')
            for i, e in enumerate(A):
                if i == len(A)-a.n:
```

```
                N.append('.')
            N.append(str(e))
        M = ''.join(N)
    else:
        B = get_complement(A)
        del B[0]
        N = []
        for i, e in enumerate(B):
            if i == len(B)-a.n:
                N.append('.')
            N.append(str(e))
        M = ''.join(N)
    return RealNum(M, a.n)

def sub_mul(a, b):
    # a > 0 and b > 0 と仮定する
    # c = a * b を計算する
    A = a.val[:]
    B = b.val[:]
    del A[0]
    del B[0]
    A.reverse()
    B.reverse()
    N = len(A) + len(B)
    C = [0] * N
    for i, e in enumerate(B):
        t = 0
        for j, f in enumerate(A):
            g = C[i+j] + e * f + t
            t = g // 10
            C[i+j] = g % 10
        C[i+len(A)] = t
    if C[len(C)-1] == 0:
        del C[len(C)-1]
    C.reverse()
    M = []
    for i, e in enumerate(C):
        if i == len(C)-2*a.n:
            M.append('.')
            M.append(str(e))
        elif i > len(C)-a.n:
            break
        else:
```

```
            M.append(str(e))
    L = ''.join(M)
    return RealNum(L, a.n)

def mul(a, b):
    if a.val[0] == 0 and b.val[0] == 0: # 正数×正数
        c = sub_mul(a, b)
    elif a.val[0] == 0 and b.val[0] == 9: # 正数×負数
        d = -b
        e = sub_mul(a, d)
        c = -e
    elif a.val[0] == 9 and b.val[0] == 0: # 負数×正数
        d = -a
        e = sub_mul(d, b)
        c = -e
    elif a.val[0] == 9 and b.val[0] == 9: # 負数×負数
        d = -a
        e = -b
        c = sub_mul(d, e)
    return c

def zeroP(a):
    A = a.val[:]
    for e in A:
        if e != 0: return False
    return True

def eq(a, b):
    c = a - b
    return zeroP(c)


def gt(a, b):
    c = b - a
    if c.val[0] == 9:
        return True
    else:
        return False
def lt(a, b):
    c = a - b
    if c.val[0] == 9:
        return True
    else:
```

```
        return False

def copy(a):
    A = a.val[:]
    del A[0]
    c = []
    for i, e in enumerate(A):
        if i == len(A)-a.n:
            c.append('.')
        c.append(str(e))
    return RealNum(''.join(c), a.n)

def single(x, m, n):
    if m > n:
        s = [str(x)]
        for i in range(m-1):
            if i == m-n-1:
                s.append('.')
            s.append('0')
        return RealNum(''.join(s), n)
    elif m <= n:
        s = ['0','.']
        for i in range(n):
            if i == n - m:
                s.append(str(x))
            else:
                s.append('0')
        r = RealNum(''.join(s), n)
        return r

def sub_div(a, b):
    # a > 0 and b > 0 と仮定する
    # q に a / b の商の数値をセット
    q = []
    m = len(a.val)-1
    d = copy(a)
    for i in range(m):
        c = 1
        p = single(c, m-i, a.n)
        if b * p > d:
            q.append(0)
            continue
        while b * p <= d:
```

```
            c += 1
            p = single(c, m-i, a.n)
        c -= 1
        p = single(c, m-i, a.n)
        q.append(c)
        d = d - b * p
    # 不要な 0 を除く
    z = 0
    for i in range(m):
        if q[i] != 0:
            z = i
            break
        elif i < m - a.n -1: continue
        elif i == m - a.n - 1:
            z = i
            break
    # 結果を文字列に変換
    Q = []
    for i, e in enumerate(q):
        if i < z: continue
        if i == m - a.n:
            Q.append('.')
        Q.append(str(e))
    return RealNum(''.join(Q), a.n)

def div(a, b):
    if a.val[0] == 0 and b.val[0] == 0: # 正数/正数
        c = sub_div(a, b)
    elif a.val[0] == 0 and b.val[0] == 9: # 正数/負数
        d = -b
        e = sub_div(a, d)
        c = -e
    elif a.val[0] == 9 and b.val[0] == 0: # 負数/正数
        d = -a
        e = sub_div(d, b)
        c = -e
    elif a.val[0] == 9 and b.val[0] == 9: # 負数/負数
        d = -a
        e = -b
        c = sub_div(d, e)
    return c
```

```
class RealNum:
    def __init__(self, x, n):
        x = x.strip()
        p = x.count('.')
        q = x.count('0')
        if x[0] == '-': # 負数
            x = x.lstrip('-')
            # 文字列をクラスにセットする数値のリストに変換する
            val = set_degit(x, n)
            self.val = get_complement(val) # 10 の補数に変換する
        elif p+q == len(x): # ゼロ
            self.val = [0, 0]
            for i in range(n):
                self.val.append(0)
        else: # 正数
            # 文字列をクラスにセットする数値のリストに変換する
            self.val = set_degit(x, n)
        self.n = n
    __add__ = add
    __neg__ = neg
    __mul__ = mul
    def __sub__(a, b):
        return a + (-b)
    def __le__(a, b):
        c = b-a
        if c.val[0] == 0:
            return True
        else:
            return False
    def __ge__(self, other):
        c = self - other
        if c.val[0] == 0:
            return True
        else:
            return False
    __gt__ = gt
    __lt__ = lt
    __eq__ = eq
    __truediv__ = div
    def print(self):
        if self.val[0] == 0: # 非負数
            val = self.val[:]
        else: # 負数
```

```
            val = get_complement(self.val) # 10 の補数に変換する
        m = len(val)
        for i, e in enumerate(val):
            if i == 0:
                if self.val[0] == 0:
                    print(' ', end=' ')
                else:
                    print('-', end=' ')
            elif i == m-self.n:
                print('.', e, end=' ')
            else:
                print(e, end=' ')
        print()
```

が全部のプログラムです。

では、さっそく円周率を計算するプログラムを作ってみましょう。

```
from pi import *

def marctan(n, d) :
    a = RealNum('0.0', 110)
    e = RealNum(str(n), 110)
    d = RealNum(str(d), 110)
    f = RealNum('0.0', 110)
    e = e / d
    a = a + e
    i = 3
    l = single(1, 2, 110)
    while e > l:
        e = e / d
        e = e / d
        I = RealNum(str(i), 110)
        f = e / I
        if i % 4 == 1:
            a = a + f
        else:
            a = a - f
        i = i + 2
        print('i=', i)
    return a

if __name__ == '__main__':

    A = marctan(16, 5); print('a=', end=' '); A.print()
```

```
    B = marctan(4, 239); print('b=', end=' '); B.print()
    pi = A - B
    print('pi=', end=' '); pi.print()
```

というプログラムを作ります。ここで、クラス RealNum を定義したプログラムを pi.py という名前で保存しています。実行すると

```
i= 5
i= 7
i= 9
 .
 .
 .
i= 157
i= 159
i= 161
a=   3 . 1 5 8 3 2 8 9 5 7 5 9 8 0 9 2 1 3 3 9 2
 0 7 9 6 2 4 3 1 1 6 6 4 4 6 9 5 1 6 1 3
 6 1 6 6 0 6 0 5 6 3 3 6 2 4 2 8 3 0 2 3
 0 4 3 8 5 6 5 4 3 5 1 9 6 5 1 9 0 0 5 7
 1 7 2 3 1 6 5 2 4 8 5 9 6 6 0 8 7 0 5 8
 0 2 0 4 9 8 8 9 1 6
i= 5
i= 7
i= 9
 .
 .
 .
i= 45
i= 47
i= 49
b=   0 . 0 1 6 7 3 6 3 0 4 0 0 8 2 9 8 8 9 5 4
 5 8 1 5 2 8 5 9 8 3 7 1 4 1 8 1 0 9 6 4
 1 9 2 2 6 1 2 3 0 5 2 7 8 0 3 3 0 8 0 7
 8 4 5 1 5 4 8 7 2 7 1 1 3 3 6 5 6 9 1 5
 7 3 0 9 5 1 3 0 4 2 3 2 5 4 4 9 1 6 3 7
 8 1 9 9 0 1 8 0 2 6 6
pi=   3 .
 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6
 2 6 4 3 3 8 3 2 7 9 5 0 2 8 8 4 1 9 7 1
 6 9 3 9 9 3 7 5 1 0 5 8 2 0 9 7 4 9 4 4
 5 9 2 3 0 7 8 1 6 4 0 6 2 8 6 2 0 8 9 9
 8 6 2 8 0 3 4 8 2 5 3 4 2 1 1 7 0 6 7 9
 8 2 1 4 8 0 8 6 5 0
```

```
>>>
```

と計算してくれます。最後の数字 0 が本当は 1 です。私のコンピュータで 20 分ぐらいかかります。だだじっとまているのは退屈ですから、

```
        print('i=', i)
```

と計算過程がわかるようにしています。数値を変えて

```
from pi import *

def marctan(n, d) :
    a = RealNum('0.0', 210)
    e = RealNum(str(n), 210)
    d = RealNum(str(d), 210)
    f = RealNum('0.0', 210)
    e = e / d
    a = a + e
    i = 3
    l = single(1, 2, 210)
    while e > l:
        e = e / d
        e = e / d
        I = RealNum(str(i), 210)
        f = e / I
        if i % 4 == 1:
            a = a + f
        else:
            a = a - f
        i = i + 2
        print('i=', i )
    return a

if __name__ == '__main__':

    A = marctan(16, 5); print('a=', end=' '); A.print()
    B = marctan(4, 239); print('b=', end=' '); B.print()
    pi = A - B
    print('pi=', end=' '); pi.print()
```

を実行すると

```
i= 5
i= 7
i= 9
 .
```

```
 .
 .
i= 301
i= 303
a=   3 . 1 5 8 3 2 8 9 5 7 5 9 8 0 9 2 1 3 3 9 2
 0 7 9 6 2 4 3 1 1 6 6 4 4 6 9 5 1 6 1 3
 6 1 6 6 0 6 0 5 6 3 3 6 2 4 2 8 3 0 2 3
 0 4 3 8 5 6 5 4 3 5 1 9 6 5 1 9 0 0 5 7
 1 7 2 3 1 6 5 2 4 8 5 9 6 6 0 8 7 0 5 8
 0 2 0 4 9 8 8 9 1 7 9 0 6 1 7 6 1 6 7 1
 3 0 3 6 7 8 1 8 8 7 0 7 8 2 9 6 9 1 9 6
 1 0 4 7 9 2 3 4 6 0 8 5 2 3 2 0 8 3 5 3
 7 1 9 5 2 0 2 5 0 3 2 0 2 4 8 9 6 0 9 8
 0 5 7 8 9 6 8 4 2 6 1 3 2 1 1 7 0 9 9 0
 8 4 9 0 6 3 0 2 4 0
i= 5
i= 7
i= 9
 .
 .
 .
i= 89
i= 91
b=   0 . 0 1 6 7 3 6 3 0 4 0 0 8 2 9 8 8 9 5 4 5
 8 1 5 2 8 5 9 8 3 7 1 4 1 8 1 0 9 6 4 1
 9 2 2 6 1 2 3 0 5 2 7 8 0 3 3 0 8 0 7 8
 4 5 1 5 4 8 7 2 7 1 1 3 3 6 5 6 9 1 5 7
 3 0 9 5 1 3 0 4 2 3 2 5 4 4 9 1 6 3 7 8
 1 9 9 0 1 8 0 2 6 6 5 7 7 9 4 5 5 0 2 4
 2 0 9 8 3 3 5 7 9 1 5 7 2 4 7 4 6 0 2 3
 5 6 8 8 5 1 5 3 3 2 3 7 1 2 0 3 3 8 5 0
 8 7 8 4 9 3 2 3 0 9 3 5 0 3 7 9 0 5 3 8
 4 1 3 2 7 3 8 9 3 6 5 8 2 8 1 3 2 7 9 4
 4 0 6 1 8 1 9 2 6 1
pi=   3 .
 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6
 2 6 4 3 3 8 3 2 7 9 5 0 2 8 8 4 1 9 7 1
 6 9 3 9 9 3 7 5 1 0 5 8 2 0 9 7 4 9 4 4
 5 9 2 3 0 7 8 1 6 4 0 6 2 8 6 2 0 8 9 9
 8 6 2 8 0 3 4 8 2 5 3 4 2 1 1 7 0 6 7 9
 8 2 1 4 8 0 8 6 5 1 3 2 8 2 3 0 6 6 4 7
 0 9 3 8 4 4 6 0 9 5 5 0 5 8 2 2 3 1 7 2
 5 3 5 9 4 0 8 1 2 8 4 8 1 1 1 7 4 5 0 2
```

```
 8 4 1 0 2 7 0 1 9 3 8 5 2 1 1 0 5 5 5 9
 6 4 4 6 2 2 9 4 8 9 5 4 9 3 0 3 8 1 9 6
 4 4 2 8 8 1 0 9 7 9
```

です。計算時間は私のコンピュータで 3 時間ぐらいです。正確な値は最後の 9 が 5 です。

いくら計算できるとはいっても、これでは時間が掛かり過ぎます。円周率の計算専用の特別なプログラミングをすれば高速に計算できますが、このようなクラスを作って置けば、色々な計算に汎用的に使えます。これはコンストラクターに問題があります。C++ では複数のコンストラクターを持てますから、柔軟にプログラミングできますが、Python ではコンストラクターが一つしか使えないので、人間が RealNum のオブジェクトを作るのを楽にしようとして、

```
X = RealNum('-7.6', 50)
X.print()
Y = RealNum('5.8', 50)
Y.print()
```

のように入力できるようにコンストラクターを決めましたが、そのために四則の計算をするたびに、頻繁に数値のリストと文字列の変換が必要でした。数値のリストと文字列の変換を駆使返していたことが実行時間が長くなった原因です。数値のリストに統一すれば、人間が RealNum のオブジェクトを作るのは面倒ですが、四則の計算のプログラミングが簡単になって、実行時間が短くなるはずです。次の節でクラス RealNum を作り直します。

3.4 クラス RealNum の見直し

前節で作ったクラス RealNum は

```
if __name__ == '__main__':
    X = RealNum('1', 50)
    X.print()
    Y = RealNum('53', 50)
    Y.print()
    Z = X / Y
    Z.print()
```

を実行して

```
  1 . 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0
  5 3 . 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 0 0 0
  0 . 0 1 8 8 6 7 9 2 0 4 5 2 8 3 0 1 8 8 6 7
   9 2 0 4 5 2 8 3 0 1 8 8 6 7 9 2 0 4 5 2
   8 3 0 1 8 8 6 7 9 2
```

のような計算をするのには便利でしたが、円周率の計算などをするのには時間が掛かり過ぎました。
クラス RealNum のコンストラクターを単純に、

```
def __init__(self, val, n):
    self.val = val
    self.n = n
```

として、クラス RealNum を作り替えてみましょう。
val には実数を構成する数字の列の（例えば、val = [0, 1, 2, 3, 0, 0, 0] ）、n には小数点以下の桁数（例えば、n=4）をセットします。これで、12.3000 を表現します。正の数には先頭に 0 を追加します。負の数　-12.3000 は 10 の補数を使って表現します。つまり、100.0000 を加えます。

$$\begin{aligned} -12.3000 + 100.0000 &= -12.3000 + 99.9999 + 0.0001 \\ &= (99.9999 - 12.3000) + 0.0001 \\ &= 87.6999 + 0.0001 \\ &= 87.7000 \end{aligned}$$

そして、[8, 7, 7, 0, 0, 0] の先頭に 9 を追加して、[9, 8, 7, 7, 0, 0, 0] で負の数であることを表すことにします。元に戻すには、100.0000 - 87.7000 を計算すれば良いです。

$$\begin{aligned} 100.0000 - 87.7000 &= 99.9999 + 0.0001 - 87.7000 \\ &= (99.9999 - 87.7000) + 0.0001 \\ &= 12.2999 + 0.0001 \\ &= 12.3000 \end{aligned}$$

です。
キーボードから前節で述べたような入力をするためには、前節で作った `set_degit(x, n)` `get_complement(val)` を利用して、関数 strRealNumt(x, n) を

```
def set_degit(x, n):
    # 文字列をクラスにセットする数値のリストに変換する
    # 整数部分の桁数と小数点以下の桁数を求める
    if x.count('.') == 0:
        I = len(x)
        m = 0
    else:
        I = x.find('.')
        m =len(x) - I - 1
    # x の数値リストを作る
    val = [0]
    for i, e in enumerate(x):
        if i < I:
```

```
            val.append(int(e))
        elif i == I:
            continue
        elif i <= I + n:
            val.append(int(e))
        else:
            break
    # 小数点以下の桁数の補正
    if n > m:
        for i in range(n-m):
            val.append(0)
    return val
def get_complement(val):
    # 10 の補数に変換する
    # val は正数か負数か
    if val[0] == 0:
        flag = True
    else:
        flag = False
    res = []
    # 右端の 0 の個数を計算する
    z = 0
    for i in range(len(val)):
        if val[len(val)-i-1] == 0:
            z += 1
        else:
            break
    # 10 の補数を取る
    for i, e in enumerate(val):
        if i == 0:
            if flag == True:
                res.append(9)
            else:
                res.append(0)
        elif i < len(val) - z - 1:
            res.append(9 - e)
        elif i == len(val) - z -1:
            res.append(10 - e)
        else:
            res.append(0)
    return res
def strRealNum(x, n):
    x = x.strip()
```

```
    p = x.count('.')
    q = x.count('0')
    if x[0] == '-': # 負数
        x = x.lstrip('-')
        # 文字列をクラスにセットする数値のリストに変換する
        val = set_degit(x, n)
        val = get_complement(val) # 10 の補数に変換する
        return RealNum(val, n)
    elif p+q == len(x): # ゼロ
        val = [0, 0]
        for i in range(n):
            val.append(0)
        return RealNum(val, n)
    else: # 正数
        # 文字列をクラスにセットする数値のリストに変換する
        val = set_degit(x, n)
        return RealNum(val, n)
class RealNum:
    def __init__(self, val, n):
        self.val = val
        self.n = n
    def print(self):
        if self.val[0] == 0:
            val = self.val[:]
        else:
            val = get_complement(self.val)
        m = len(val)
        for i, e in enumerate(val):
            if i == 0:
                if self.val[0] == 0:
                    print(' ', end=' ')
                else:
                    print('-', end=' ')
            elif i == m-self.n:
                print('.', e, end=' ')
            else:
                print(e, end=' ')
        print()
```

と定義して、

```
if __name__ == '__main__':
    a = strRealNum('12.3', 4)
    a.print()
```

```
    a = strRealNum('-12.3', 4)
    a.print()
    a = strRealNum('0', 4)
    a.print()
```

を実行すれば

```
  1 2 . 3 0 0 0
- 1 2 . 3 0 0 0
  0 . 0 0 0 0
```

です。

上で述べたように 12.3000 を表現するのに self.val = [0, 1, 2, 3, 0, 0, 0] 、n=4 や -12.3000 を表現するのに self.val = [9, 8, 7, 7, 0, 0, 0] 、n=4 とセットするほかに、リストを逆にして 12.3000 を表現するのに self.val = [0, 0, 0, 3, 2, 1, 0] 、n=4 や -12.3000 を表現するのに self.val = [0, 0, 0, 7, 7, 8, 9] 、n=4 とセットすることもできます。前節でのプログラミングで見たように、割り算には前者が、足し算、引き算、掛け算では後者が便利ですが、直感的に分かりやすい前者を使うことにします。

前節のプログラミングが理解できていたら、以下のプログラミングで必要な技能はすべて経験済みですから、以下の解説を読まずに自分でクラス RealNum のプログラミングをやってみることをお勧めします。

足し算を定義しましょう。前節の関数 add(a, b) を

```
def add(a, b):
    A = a.val[:]
    B = b.val[:]
    c = []
    A.reverse()
    B.reverse()
    # 整数部分の桁数を揃える
    if len(A) > len(B):
        if B[len(B)-1] == 0:
            for i in range(len(A)-len(B)):
                B.append(0)
        elif B[len(B)-1] == 9:
            for i in range(len(A)-len(B)):
                B.append(9)
    elif len(A) < len(B):
        if A[len(A)-1] == 0:
            for i in range(len(B)-len(A)):
                A.append(0)
        elif A[len(A)-1] == 9:
            for i in range(len(B)-len(A)):
                A.append(9)
    # 和の計算
```

```
    t = 0
    for i in range(len(A)):
        x = A[i] + B[i] + t
        c.append(x % 10)
        t = x // 10
    # 計算結果の整理
    if c[len(c)-1] == 1:
        # 正数、繰り上がり
        c.append(0)
    elif c[len(c)-1] == 0:
        # 正数又は0、余分な0を除く
        p = len(c)
        del c[p-1]
        for i in range(len(c)-1, a.n, -1):
            if c[i] == 0:
                p -= 1
            else:
                break
        del c[p-1:]
        c.append(0)
    elif c[len(c)-1] == 8:
        # 負数、繰り上がり
        c.append('9')
    elif c[len(c)-1] == 9:
        # 負数、余分な9を除く
        p = len(c)
        for i in range(len(c)-1, a.n, -1):
            if c[i] == 9 and c[i-1] == 9:
                p -= 1
            else:
                break
        del c[p:]
    # オブジェクトを作成
    c.reverse()
    return RealNum(c, a.n)
```

と修正し、クラス RealNum に

```
    __add__ = add
```

を追加し、

```
if __name__ == '__main__':
    a = strRealNum('12.3', 4); a.print()
    b = strRealNum('4.56', 4); b.print()
```

```
    c = a + b; c.print()
    a = strRealNum('42.34', 4); a.print()
    b = strRealNum('75.6', 4); b.print()
    c = a + b; c.print()
    a = strRealNum('-62.5', 4); a.print()
    b = strRealNum('13.45', 4); b.print()
    c = a + b; c.print()
    a = strRealNum('-12.5', 4); a.print()
    b = strRealNum('-26.63', 4); b.print()
    c = a + b; c.print()
    a = strRealNum('-62.5', 4); a.print()
    b = strRealNum('-57.77', 4); b.print()
    c = a + b; c.print()
    a = strRealNum('-62.5', 4); a.print()
    b = strRealNum('84.67', 4); b.print()
    c = a + b; c.print()
    a = strRealNum('12.3', 4); a.print()
    b = strRealNum('-12.3', 4); b.print()
    c = a + b; c.print()
```

を実行すると

```
  1 2 . 3 0 0 0
  4 . 5 6 0 0
  1 6 . 8 6 0 0
  4 2 . 3 4 0 0
  7 5 . 6 0 0 0
  1 1 7 . 9 4 0 0
- 6 2 . 5 0 0 0
  1 3 . 4 5 0 0
- 4 9 . 0 5 0 0
- 1 2 . 5 0 0 0
- 2 6 . 6 3 0 0
- 3 9 . 1 3 0 0
- 6 2 . 5 0 0 0
- 5 7 . 7 7 0 0
- 1 2 0 . 2 7 0 0
- 6 2 . 5 0 0 0
  8 4 . 6 7 0 0
  2 2 . 1 7 0 0
  1 2 . 3 0 0 0
- 1 2 . 3 0 0 0
  0 . 0 0 0 0
```

となります。桁上がりのあるなしを含めて、すべての組み合わせをチェックします。

次に、単項演算子 - を定義し、引き算ができるようにします。前節の neg(a) が

```
def neg(a):
    if zeroP(a.val):
        return a
    A = a.val[:]
    B = get_complement(A)
    return RealNum(B, a.n)
```

と簡単になります。関数 get_complement(A) を呼び出すだけです。クラス RealNum に

```
    __neg__ = neg
    def __sub__(a, b):
        return a + (-b)
```

を追加します。

```
if __name__ == '__main__':
    a = strRealNum('0', 4); a.print()
    c = -a; c.print()
    a = strRealNum('62.5', 4); a.print()
    c = -a; c.print()
    a = strRealNum('-12.3', 4); a.print()
    c = -a; c.print()
    a = strRealNum('62.5', 4); a.print()
    b = strRealNum('84.67', 4); b.print()
    c = a - b; c.print()
    a = strRealNum('-12.3', 4); a.print()
    b = strRealNum('-12.3', 4); b.print()
    c = a - b; c.print()
```

を実行すると

```
  0 . 0 0 0 0
  0 . 0 0 0 0
  6 2 . 5 0 0 0
- 6 2 . 5 0 0 0
- 1 2 . 3 0 0 0
  1 2 . 3 0 0 0
  6 2 . 5 0 0 0
  8 4 . 6 7 0 0
- 2 2 . 1 7 0 0
- 1 2 . 3 0 0 0
- 1 2 . 3 0 0 0
  0 . 0 0 0 0
```

です。

次に掛け算を可能にします。sub_mul(a, b) を修正し、

```
def sub_mul(a, b):
    # a > 0 and b > 0 と仮定する
    # c = a * b を計算する
    A = a.val[:]
    B = b.val[:]
    del A[0]
    del B[0]
    A.reverse()
    B.reverse()
    N = len(A) + len(B)
    C = [0] * N
    for i, e in enumerate(B):
        t = 0
        for j, f in enumerate(A):
            g = C[i+j] + e * f + t
            t = g // 10
            C[i+j] = g % 10
        C[i+len(A)] = t
    if C[i+len(A)] != 0:
        C.append(0)
    del C[:a.n]
    C.reverse()
    return RealNum(C, a.n)

def mul(a, b):
    if a.val[0] == 0 and b.val[0] == 0: # 正数×正数
        c = sub_mul(a, b)
    elif a.val[0] == 0 and b.val[0] == 9: # 正数×負数
        d = -b
        e = sub_mul(a, d)
        c = -e
    elif a.val[0] == 9 and b.val[0] == 0: # 負数×正数
        d = -a
        e = sub_mul(d, b)
        c = -e
    elif a.val[0] == 9 and b.val[0] == 9: # 負数×負数
        d = -a
        e = -b
        c = sub_mul(d, e)
    return c
```

と定義し、クラス RealNum に

```
    __mul__ = mul
```

を追加し、

```
    a = strRealNum('6.5', 4); a.print()
    b = strRealNum('84', 4); b.print()
    c = a * b; c.print()
    a = strRealNum('-12.', 4); a.print()
    b = strRealNum('-3.3', 4); b.print()
    c = a * b; c.print()
```

を実行すると

```
  6 . 5 0 0 0
  8 4 . 0 0 0 0
  5 4 6 . 0 0 0 0
- 1 2 . 0 0 0 0
- 3 . 3 0 0 0
  3 9 . 6 0 0 0
```

です。

割り算ができるようにするために、比較演算子 == と <= と >= と < と > をオーバーロードしておきましょう。

```
def eq(a, b):
    c = a - b
    return zeroP(c.val)


def gt(a, b):
    c = b - a
    if c.val[0] == 9:
        return True
    else:
        return False
def lt(a, b):
    c = a - b
    if c.val[0] == 9:
        return True
    else:
        return False
```

と定義し、クラス RealNum に

```
    __gt__ = gt
    __lt__ = lt
    __eq__ = eq
```

```
    def __le__(a, b):
        c = b-a
        if c.val[0] == 0:
            return True
        else:
            return False
    def __ge__(self, other):
        c = self - other
        if c.val[0] == 0:
            return True
        else:
            return False
```

を追加します。

割り算する関数も前節の関数を修正すれば良いです。

```
def copy(a):
    A = a.val[:]
    return RealNum(A, a.n)
def single(x, m, n):
    if m > n:
        s = [0, x]
        for i in range(m-1):
            s.append(0)
        return RealNum(s, n)
    elif m <= n:
        s = [0]
        for i in range(n):
            if i == n - m:
                s.append(x)
            else:
                s.append(0)
        return RealNum(s, n)
def sub_div(a, b):
    # a > 0 and b > 0 と仮定する
    # a / b を計算する
    q = []
    m = len(a.val)-1
    d = copy(a)
    for i in range(m):
        c = 1
        p = single(c, m-i, a.n)
        if b * p > d:
            q.append(0)
```

```
            continue
        while c < 10 and b * p <= d:
            c += 1
            p = single(c, m-i, a.n)
        c -= 1
        p = single(c, m-i, a.n)
        q.append(c)
        d = d - b * p
    # 必要ない 0 の範囲を計算する
     z = 0
    for i in range(m):
        if q[i] != 0:
            z = -1 # 0 を追加する必要がある
        else:
            if i < m - a.n - 1 and q[i] == 0 and q[i+1] == 0:
                continue
            elif i < m - a.n - 1 and q[i] == 0 and q[i+1] != 0:
                z = i
                break
            elif i == m - a.n - 1:
                z = i-1
                break
    if z > 0:
        del q[:z]
    elif z < 0:
        q.reverse()
        q.append(0)
        q.reverse()
    return RealNum(q, a.n)
def div(a, b):
    if a.val[0] == 0 and b.val[0] == 0: # 正数/正数
        c = sub_div(a, b)
    elif a.val[0] == 0 and b.val[0] == 9: # 正数/負数
        d = -b
        e = sub_div(a, d)
        c = -e
    elif a.val[0] == 9 and b.val[0] == 0: # 負数/正数
        d = -a
        e = sub_div(d, b)
        c = -e
    elif a.val[0] == 9 and b.val[0] == 9: # 負数/負数
        d = -a
        e = -b
```

```
        c = sub_div(d, e)
    return c
```

と定義し、クラス RealNum に

```
    __truediv__ = div
```

を追加し、

```
    a = strRealNum('5', 4); a.print()
    b = strRealNum('4', 4); b.print()
    c = a / b; c.print()
    a = strRealNum('50', 4); a.print()
    b = strRealNum('3', 4); b.print()
    c = a / b; c.print()
    a = strRealNum('50', 4); a.print()
    b = strRealNum('8', 4); b.print()
    c = a / b; c.print()
    a = strRealNum('0.50', 4); a.print()
    b = strRealNum('3', 4); b.print()
    c = a / b; c.print()
    a = strRealNum('500', 4); a.print()
    b = strRealNum('80', 4); b.print()
    c = a / b; c.print()
```

を実行すると

```
 5 . 0 0 0 0
 4 . 0 0 0 0
 1 . 2 5 0 0
 5 0 . 0 0 0 0
 3 . 0 0 0 0
 1 6 . 6 6 6 6
 5 0 . 0 0 0 0
 8 . 0 0 0 0
 6 . 2 5 0 0
 0 . 5 0 0 0
 3 . 0 0 0 0
 0 . 1 6 6 6
 5 0 0 . 0 0 0 0
 8 0 . 0 0 0 0
 6 . 2 5 0 0
```

です。

このクラス RealNum を使って、円周率を計算してみましょう。

```
from RealNum import *

def marctan(n, d) :
    a = strRealNum('0.0', 110)
    e = strRealNum(str(n), 110)
    d = strRealNum(str(d), 110)
    f = strRealNum('0.0', 110)
    e = e / d
    a = a + e
    i = 3
    l = single(1, 2, 110)
    while e > l:
        e = e / d
        e = e / d
        I = strRealNum(str(i), 110)
        f = e / I
        if i % 4 == 1:
            a = a + f
        else:
            a = a - f
        i = i + 2
        print('i=', i)
    return a

if __name__ == '__main__':

    A = marctan(16, 5); print('a=', end=' '); A.print()
    B = marctan(4, 239); print('b=', end=' '); B.print()
    pi = A - B
    print('pi=', end=' '); pi.print()
```

というプログラムを作ります。ここで、クラス RealNum を定義したプログラムを RealNumpy という名前で保存しています。実行すると

```
i= 5
i= 7
i= 9
 .
 .
 .
i= 157
i= 159
i= 161
a=   3 . 1 5 8 3 2 8 9 5 7 5 9 8 0 9 2 1 3 3 9 2
```

```
 0 7 9 6 2 4 3 1 1 6 6 4 4 6 9 5 1 6 1 3
 6 1 6 6 0 6 0 5 6 3 3 6 2 4 2 8 3 0 2 3
 0 4 3 8 5 6 5 4 3 5 1 9 6 5 1 9 0 0 5 7
 1 7 2 3 1 6 5 2 4 8 5 9 6 6 0 8 7 0 5 8
 0 2 0 4 9 8 8 9 1 6
i= 5
i= 7
i= 9
 .
 .
 .
 i= 45
i= 47
i= 49
b=   0 . 0 1 6 7 3 6 3 0 4 0 0 8 2 9 8 8 9 5 4 5
 8 1 5 2 8 5 9 8 3 7 1 4 1 8 1 0 9 6 4 1
 9 2 2 6 1 2 3 0 5 2 7 8 0 3 3 0 8 0 7 8
 4 5 1 5 4 8 7 2 7 1 1 3 3 6 5 6 9 1 5 7
 3 0 9 5 1 3 0 4 2 3 2 5 4 4 9 1 6 3 7 8
 1 9 9 0 1 8 0 2 6 6
pi=   3 .
 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6
 2 6 4 3 3 8 3 2 7 9 5 0 2 8 8 4 1 9 7 1
 6 9 3 9 9 3 7 5 1 0 5 8 2 0 9 7 4 9 4 4
 5 9 2 3 0 7 8 1 6 4 0 6 2 8 6 2 0 8 9 9
 8 6 2 8 0 3 4 8 2 5 3 4 2 1 1 7 0 6 7 9
 8 2 1 4 8 0 8 6 5 0
```

です。私のコンピュータでは 12 分ぐらいで計算しました。2 倍弱速くなりました。円周率を計算するには、割り算は int の範囲内で割ることしか必要ないので、汎用的な割り算が必要ないことです。例えば、クラス RealNum を int で割るプログラムを作り、それを使えばもっと早く計算してくれるはずです。やってみましょう。

```
def intdiv(a, k):
    # a > 0 and k > 0 で k は正数と仮定する
    # a / i を計算する
    q = []
    m = len(a.val)-1
    t = 0
    for i in range(len(a.val)):
        x = t * 10 + a.val[i]
        q.append(x // k)
        t = x % k
    # 必要ない 0 の範囲を計算する
```

```
    # q.reverse()
    print('a.val=', a.val)
    print('q=', q)
    print('m=', m)
    z = 0
    for i in range(m):
        if q[i] != 0:
            z = -1 # 0 を追加する必要がある
        else:
            if i < m - a.n - 1 and q[i+1] == 0:
                continue
            elif i < m - a.n - 1 and q[i+1] != 0:
                z = i
                break
            elif i == m - a.n - 1:
                z = i
                break
    print('z=', z)
    if z > 0:
        del q[:z]
        print('2: q=', q)
    elif z < 0:
        q.reverse()
        q.append(0)
        q.reverse()
    return RealNum(q, a.n)
```

と定義し、クラス RealNum に

```
    __floordiv__ = intdiv
```

を追加すれば、RealNum // int の形で割り算ができるようになり、円周率を計算するプログラムを

```
from RealNum import *

def marctan(n, d) :
    a = strRealNum('0.0', 1100)
    e = strRealNum(str(n), 1100)
    f = strRealNum('0.0', 1100)
    e = e // d
    a = a + e
    i = 3
    l = single(1, 2, 1100)
    while e > l:
        e = e // d
```

```
        e = e // d
        f = e // i
        if i % 4 == 1:
            a = a + f
        else:
            a = a - f
        i = i + 2
        print('i=', i)
    return a

if __name__ == '__main__':

    A = marctan(16, 5); print('a=', end=' '); A.print()
    B = marctan(4, 239); print('b=', end=' '); B.print()
    pi = A - B
    print('pi=', end=' '); pi.print()
```

と修正してを実行すると

```
i= 5
i= 7
i= 9
 .
 .
 .
i= 1573
i= 1575
i= 1577
a=   3 . 1 5 8 3 2 8 9 5 7 5 9 8 0 9 2 1 3 3 9 2
 0 7 9 6 2 4 3 1 1 6 6 4 4 6 9 5 1 6 1 3
 6 1 6 6 0 6 0 5 6 3 3 6 2 4 2 8 3 0 2 3
 0 4 3 8 5 6 5 4 3 5 1 9 6 5 1 9 0 0 5 7
 1 7 2 3 1 6 5 2 4 8 5 9 6 6 0 8 7 0 5 8
 0 2 0 4 9 8 8 9 1 7 9 0 6 1 7 6 1 6 7 1
 3 0 3 6 7 8 1 8 8 7 0 7 8 2 9 6 9 1 9 6
 1 0 4 7 9 2 3 4 6 0 8 5 2 3 2 0 8 3 5 3
 7 1 9 5 2 0 2 5 0 3 2 0 2 4 8 9 6 0 9 8
 0 5 7 8 9 6 8 4 2 6 1 3 2 1 1 7 0 9 9 0
 8 4 9 0 6 3 0 2 3 4 8 8 9 9 0 3 2 4 5 4
 4 9 3 2 1 4 8 9 7 9 8 8 0 2 5 8 1 2 7 5
 7 0 2 3 8 9 4 3 5 3 8 1 1 7 3 6 6 0 5 1
 4 8 7 3 7 3 4 6 8 2 6 3 7 2 9 6 8 4 0 7
 5 6 8 6 3 4 6 5 1 6 1 0 4 0 7 8 2 2 0 5
 0 4 8 4 9 5 0 1 4 8 6 4 9 2 6 8 9 8 7 1
```

```
 3 4 9 4 7 0 8 5 6 2 8 9 0 1 8 3 8 7 1 5
 1 6 0 3 1 0 4 2 7 3 3 6 9 7 5 1 9 2 4 3
 7 8 9 8 7 7 9 8 1 6 1 3 4 4 1 2 9 9 7 8
 8 5 6 9 8 5 2 8 1 4 9 2 7 7 1 6 0 2 2 4
 3 7 1 9 4 6 8 0 8 7 8 7 3 7 6 4 2 7 2 7
 3 1 9 0 0 6 1 0 4 7 5 9 7 4 2 5 2 7 9 6
 6 3 7 6 0 3 2 1 2 0 3 5 7 9 1 4 9 4 0 7
 3 8 2 2 7 6 4 5 3 6 6 7 0 9 2 0 9 1 5 5
 7 2 5 4 6 2 1 5 4 3 6 3 9 0 6 1 6 4 6 5
 5 5 2 8 7 4 4 1 9 5 9 0 1 2 8 4 3 1 6 0
 9 6 8 2 9 4 0 4 1 1 8 7 5 3 4 0 4 5 3 7
 3 6 5 0 4 3 9 4 3 5 9 8 9 2 9 4 6 3 3 4
 0 2 1 4 1 1 6 9 7 9 9 8 4 5 3 1 2 4 8 2
 0 2 8 8 8 1 6 1 0 7 2 3 5 7 3 0 0 1 2 0
 0 8 1 5 0 7 9 1 7 5 3 6 4 2 7 4 5 9 6 0
 7 8 3 0 5 2 6 7 6 1 8 2 4 7 4 8 7 9 2 0
 9 9 7 8 5 5 5 0 8 5 2 5 9 6 8 9 3 2 6 0
 5 9 5 2 8 5 7 6 4 8 5 5 3 1 3 2 2 5 7 0
 6 6 9 0 1 9 8 9 0 0 1 0 6 0 2 7 8 9 0 2
 5 4 3 8 8 1 6 9 1 8 6 7 2 5 6 7 2 6 3 0
 4 7 4 9 0 1 7 9 5 8 5 7 7 2 2 8 9 2 9 6
 0 5 7 9 4 2 0 1 5 8 8 1 5 3 4 2 5 1 4 6
 6 6 3 7 6 2 3 2 4 0 9 1 3 0 8 7 9 2 7 8
 0 9 6 0 6 9 0 1 2 7 2 7 4 6 9 4 2 2 7 6
 6 4 5 9 0 1 2 9 4 9 2 4 5 6 6 1 3 0 0 0
 3 4 8 3 0 2 7 9 7 4 6 6 0 0 8 9 3 9 9 1
 0 3 1 1 6 5 9 5 0 2 0 0 2 2 5 8 2 0 7 6
 5 4 8 9 0 7 3 2 9 6 5 3 1 7 7 9 6 3 7 3
 2 9 4 0 0 7 7 3 6 1 1 5 6 8 6 7 3 5 5 0
 6 4 8 5 9 9 3 1 0 9 1 9 1 1 6 0 9 0 4 4
 9 4 3 8 7 3 9 5 1 0 1 6 6 6 3 1 4 3 2 1
 3 3 8 9 8 7 5 2 8 2 9 6 3 0 4 0 4 2 5 9
 6 4 3 9 1 5 1 0 2 9 2 0 6 5 4 3 3 9 0 6
 5 3 7 6 3 6 0 7 9 1 1 8 9 9 8 8 6 8 3 5
 6 4 3 3 8 5 6 6 2 7 0 1 3 0 1 6 6 4 7 7
 4 1 6 7 7 6 8 2 8 3 4 0 7 9 4 2 8 5 4 2
 4 4 5 5 8 5 6 6 6 0 1 8 2 4 1 6 6 2 4 6
 6 6 4 2 0 8 8 6 9 9 1 0 7 0 2 2 1 2 2 2
 0 5 5 0 0 4 3 8 3 1 5 7 1 8 2 9 6 6 6 4
i= 5
i= 7
i= 9
 .
```

```
 .
 .
i= 461
i= 463
i= 465
b=   0 . 0 1 6 7 3 6 3 0 4 0 0 8 2 9 8 8 9 5 4 5
 8 1 5 2 8 5 9 8 3 7 1 4 1 8 1 0 9 6 4 1
 9 2 2 6 1 2 3 0 5 2 7 8 0 3 3 0 8 0 7 8
 4 5 1 5 4 8 7 2 7 1 1 3 3 6 5 6 9 1 5 7
 3 0 9 5 1 3 0 4 2 3 2 5 4 4 9 1 6 3 7 8
 1 9 9 0 1 8 0 2 6 6 5 7 7 9 4 5 5 0 2 4
 2 0 9 8 3 3 5 7 9 1 5 7 2 4 7 4 6 0 2 3
 5 6 8 8 5 1 5 3 3 2 3 7 1 2 0 3 3 8 5 0
 8 7 8 4 9 3 2 3 0 9 3 5 0 3 7 9 0 5 3 8
 4 1 3 2 7 3 8 9 3 6 5 8 2 8 1 3 2 7 9 4
 4 0 6 1 8 1 9 2 5 9 2 2 3 9 6 9 7 9 9 3
 2 0 8 4 5 8 4 1 5 6 5 0 1 5 7 9 8 1 1 0
 4 3 1 1 8 7 5 2 6 2 3 5 5 2 5 0 9 3 5 9
 1 4 1 3 3 8 6 0 7 2 1 8 2 9 7 0 1 9 2 5
 4 3 4 6 9 8 5 7 9 0 0 7 9 1 6 4 0 9 3 1
 3 2 3 9 0 8 0 0 8 2 5 8 6 1 1 3 1 0 5 3
 8 6 1 3 1 8 7 6 4 1 9 2 7 3 5 4 6 1 7 4
 2 4 3 1 5 6 7 8 3 6 5 8 0 4 9 2 8 8 8 3
 7 7 8 5 4 7 4 5 1 0 6 4 6 2 0 8 3 3 2 6
 7 1 8 5 7 0 5 8 6 2 9 8 6 2 0 4 4 1 3 0
 0 4 1 3 7 4 1 0 5 1 2 9 7 8 0 5 0 7 7 4
 2 2 6 8 1 9 9 8 7 3 7 7 8 0 9 9 1 6 1 7
 3 2 7 0 9 1 3 5 7 2 2 8 3 4 5 2 5 6 0 7
 7 5 4 7 8 0 7 8 0 1 4 8 2 3 4 5 6 4 3 0
 8 3 4 2 3 4 2 1 6 1 8 0 8 9 4 2 1 5 5 2
 5 6 9 5 0 7 0 8 3 3 4 6 0 6 2 7 6 7 3 0
 1 0 8 0 8 0 0 9 1 7 2 3 5 8 1 7 9 8 0 0
 1 7 4 3 4 1 7 6 3 7 3 7 9 8 5 7 6 0 5 6
 9 6 7 4 8 9 9 8 0 3 6 9 1 3 5 4 4 9 5 8
 1 8 2 1 3 3 4 2 6 0 4 6 8 7 8 9 4 9 8 8
 0 8 0 9 3 9 7 9 0 3 9 1 1 6 3 8 9 8 7 8
 0 0 4 4 7 5 5 4 1 9 0 6 6 9 5 9 1 8 2 9
 2 6 1 4 8 3 7 2 1 3 1 1 2 8 4 5 2 3 5 9
 3 7 0 3 3 2 3 3 4 7 0 8 7 6 3 6 4 0 3 3
 5 6 3 9 4 0 6 6 2 0 4 1 6 7 6 8 9 6 6 7
 1 2 3 6 8 2 1 3 0 7 4 5 9 6 6 5 0 6 6 9
 6 1 0 8 6 7 3 7 7 6 9 7 9 0 9 2 6 3 1 8
 5 8 0 8 1 1 0 1 9 8 2 9 6 6 3 5 3 0 3 3
```

```
 1 6 3 7 6 2 3 4 0 3 6 1 5 2 8 2 9 3 2 7
 0 3 6 3 3 7 2 7 9 9 1 1 3 7 3 1 0 4 1 7
 1 4 3 4 5 5 3 4 9 3 8 9 8 7 5 2 9 9 7 3
 9 2 3 0 7 9 7 1 4 9 3 2 5 6 2 0 8 9 5 5
 7 6 9 2 3 4 7 6 2 0 2 9 2 1 5 8 1 7 6 2
 7 6 5 0 3 2 0 4 0 9 9 4 4 2 4 6 4 2 8 9
 4 7 9 8 0 1 5 6 4 3 3 8 9 9 5 2 6 2 4 7
 0 5 0 3 4 5 8 2 0 4 9 0 3 6 0 6 2 1 7 1
 8 2 7 9 1 7 6 6 4 6 2 8 4 2 7 7 6 4 4 5
 4 0 1 4 6 7 9 5 0 4 7 7 7 2 6 2 3 7 2 7
 4 7 2 6 8 8 2 9 6 3 0 7 6 5 2 4 1 1 1 8
 8 7 6 5 1 6 4 8 8 1 9 7 3 5 6 8 4 8 4 6
 2 6 2 4 3 3 0 9 0 6 9 0 6 4 6 8 0 6 1 4
 1 3 7 9 1 0 8 9 2 1 8 7 4 1 2 4 5 7 4 5
 6 2 2 5 5 5 4 7 0 8 1 4 7 1 1 4 7 7 1 6
 9 7 4 2 5 1 1 3 3 6 8 4 7 0 8 0 7 3 3 0
 8 0 5 2 8 2 6 0 7 8 7 3 7 0 3 8 3 5 0 2
pi=   3 . 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6
 2 6 4 3 3 8 3 2 7 9 5 0 2 8 8 4 1 9 7 1
 6 9 3 9 9 3 7 5 1 0 5 8 2 0 9 7 4 9 4 4
 5 9 2 3 0 7 8 1 6 4 0 6 2 8 6 2 0 8 9 9
 8 6 2 8 0 3 4 8 2 5 3 4 2 1 1 7 0 6 7 9
 8 2 1 4 8 0 8 6 5 1 3 2 8 2 3 0 6 6 4 7
 0 9 3 8 4 4 6 0 9 5 5 0 5 8 2 2 3 1 7 2
 5 3 5 9 4 0 8 1 2 8 4 8 1 1 1 7 4 5 0 2
 8 4 1 0 2 7 0 1 9 3 8 5 2 1 1 0 5 5 5 9
 6 4 4 6 2 2 9 4 8 9 5 4 9 3 0 3 8 1 9 6
 4 4 2 8 8 1 0 9 7 5 6 6 5 9 3 3 4 4 6 1
 2 8 4 7 5 6 4 8 2 3 3 7 8 6 7 8 3 1 6 5
 2 7 1 2 0 1 9 0 9 1 4 5 6 4 8 5 6 6 9 2
 3 4 6 0 3 4 8 6 1 0 4 5 4 3 2 6 6 4 8 2
 1 3 3 9 3 6 0 7 2 6 0 2 4 9 1 4 1 2 7 3
 7 2 4 5 8 7 0 0 6 6 0 6 3 1 5 5 8 8 1 7
 4 8 8 1 5 2 0 9 2 0 9 6 2 8 2 9 2 5 4 0
 9 1 7 1 5 3 6 4 3 6 7 8 9 2 5 9 0 3 6 0
 0 1 1 3 3 0 5 3 0 5 4 8 8 2 0 4 6 6 5 2
 1 3 8 4 1 4 6 9 5 1 9 4 1 5 1 1 6 0 9 4
 3 3 0 5 7 2 7 0 3 6 5 7 5 9 5 9 1 9 5 3
 0 9 2 1 8 6 1 1 7 3 8 1 9 3 2 6 1 1 7 9
 3 1 0 5 1 1 8 5 4 8 0 7 4 4 6 2 3 7 9 9
 6 2 7 4 9 5 6 7 3 5 1 8 8 5 7 5 2 7 2 4
 8 9 1 2 2 7 9 3 8 1 8 3 0 1 1 9 4 9 1 2
 9 8 3 3 6 7 3 3 6 2 4 4 0 6 5 6 6 4 3 0
```

```
 8 6 0 2 1 3 9 4 9 4 6 3 9 5 2 2 4 7 3 7
 1 9 0 7 0 2 1 7 9 8 6 0 9 4 3 7 0 2 7 7
 0 5 3 9 2 1 7 1 7 6 2 9 3 1 7 6 7 5 2 3
 8 4 6 7 4 8 1 8 4 6 7 6 6 9 4 0 5 1 3 2
 0 0 0 5 6 8 1 2 7 1 4 5 2 6 3 5 6 0 8 2
 7 7 8 5 7 7 1 3 4 2 7 5 7 7 8 9 6 0 9 1
 7 3 6 3 7 1 7 8 7 2 1 4 6 8 4 4 0 9 0 1
 2 2 4 9 5 3 4 3 0 1 4 6 5 4 9 5 8 5 3 7
 1 0 5 0 7 9 2 2 7 9 6 8 9 2 5 8 9 2 3 5
 4 2 0 1 9 9 5 6 1 1 2 1 2 9 0 2 1 9 6 0
 8 6 4 0 3 4 4 1 8 1 5 9 8 1 3 6 2 9 7 7
 4 7 7 1 3 0 9 9 6 0 5 1 8 7 0 7 2 1 1 3
 4 9 9 9 9 9 9 8 3 7 2 9 7 8 0 4 9 9 5 1
 0 5 9 7 3 1 7 3 2 8 1 6 0 9 6 3 1 8 5 9
 5 0 2 4 4 5 9 4 5 5 3 4 6 9 0 8 3 0 2 6
 4 2 5 2 2 3 0 8 2 5 3 3 4 4 6 8 5 0 3 5
 2 6 1 9 3 1 1 8 8 1 7 1 0 1 0 0 0 3 1 3
 7 8 3 8 7 5 2 8 8 6 5 8 7 5 3 3 2 0 8 3
 8 1 4 2 0 6 1 7 1 7 7 6 6 9 1 4 7 3 0 3
 5 9 8 2 5 3 4 9 0 4 2 8 7 5 5 4 6 8 7 3
 1 1 5 9 5 6 2 8 6 3 8 8 2 3 5 3 7 8 7 5
 9 3 7 5 1 9 5 7 7 8 1 8 5 7 7 8 0 5 3 2
 1 7 1 2 2 6 8 0 6 6 1 3 0 0 1 9 2 7 8 7
 6 6 1 1 1 9 5 9 0 9 2 1 6 4 2 0 1 9 8 9
 3 8 0 9 5 2 5 7 2 0 1 0 6 5 4 8 5 8 6 3
 2 7 8 8 6 5 9 3 6 1 5 3 3 8 1 8 2 7 9 6
 8 2 3 0 3 0 1 9 5 2 0 3 5 3 0 1 8 5 2 9
 6 8 9 9 5 7 7 3 6 2 2 5 9 9 4 1 3 8 9 1
 2 4 9 7 2 1 7 7 5 2 8 3 4 7 9 1 3 1 6 2
```

の結果を私のコンピュータで 50 秒で実行するようになりました。1100 桁まで計算すれば、正確な誤差の解析をしなくても、1000 桁ぐらいは正しいはずです。このように思いついたことを色々プログラミングして、実験していると自然にプログラミングが出来るようになります。

```
from RealNum import *
```

は、クラス RealNum を定義したプログラム

```
def zeroP(val):
    for e in val:
        if e != 0: return False
    return True


def set_degit(x, n):
    # 文字列をクラスにセットする数値のリストに変換する
```

```
    # 整数部分の桁数と小数点以下の桁数を求める
    if x.count('.') == 0:
        I = len(x)
        m = 0
    else:
        I = x.find('.')
        m =len(x) - I - 1
    # x の数値リストを作る
    val = [0]
    for i, e in enumerate(x):
        if i < I:
            val.append(int(e))
        elif i == I:
            continue
        elif i <= I + n:
            val.append(int(e))
        else:
            break
    # 小数点以下の桁数の補正
    if n > m:
        for i in range(n-m):
            val.append(0)
    return val

def get_complement(val):
    # 10 の補数に変換する
    # val は正数か負数か
    if val[0] == 0:
        flag = True
    else:
        flag = False
    res = []
    # 右端の 0 の個数を計算する
    z = 0
    for i in range(len(val)):
        if val[len(val)-i-1] == 0:
            z += 1
        else:
            break
    # 10 の補数を取る
    for i, e in enumerate(val):
        if i == 0:
            if flag == True:
```

```
                res.append(9)
            else:
                res.append(0)
        elif i < len(val) - z - 1:
            res.append(9 - e)
        elif i == len(val) - z -1:
            res.append(10 - e)
        else:
            res.append(0)
    return res

def strRealNum(x, n):
    x = x.strip()
    p = x.count('.')
    q = x.count('0')
    if x[0] == '-': # 負数
        x = x.lstrip('-')
        # 文字列をクラスにセットする数値のリストに変換する
        val = set_degit(x, n)
        val = get_complement(val) # 10 の補数に変換する
        return RealNum(val, n)
    elif p+q == len(x): # ゼロ
        val = [0, 0]
        for i in range(n):
            val.append(0)
        return RealNum(val, n)
    else: # 正数
        # 文字列をクラスにセットする数値のリストに変換する
        val = set_degit(x, n)
        return RealNum(val, n)

def add(a, b):
    A = a.val[:]
    B = b.val[:]
    c = []
    A.reverse()
    B.reverse()
    # 整数部分の桁数を揃える
    if len(A) > len(B):
        if B[len(B)-1] == 0:
            for i in range(len(A)-len(B)):
                B.append(0)
        elif B[len(B)-1] == 9:
```

```
        for i in range(len(A)-len(B)):
            B.append(9)
elif len(A) < len(B):
    if A[len(A)-1] == 0:
        for i in range(len(B)-len(A)):
            A.append(0)
    elif A[len(A)-1] == 9:
        for i in range(len(B)-len(A)):
            A.append(9)
# 和の計算
t = 0
for i in range(len(A)):
    x = A[i] + B[i] + t
    c.append(x % 10)
    t = x // 10
# 計算結果の整理
if c[len(c)-1] == 1:
    # 正数、繰り上がり
    c.append(0)
    #C = convert_pos(c, a.n) # 正数のリストを文字列に変換する
elif c[len(c)-1] == 0:
    # 正数又は0、余分な0を除く
    p = len(c)
    del c[p-1]
    for i in range(len(c)-1, a.n, -1):
        if c[i] == 0:
            p -= 1
        else:
            break
    del c[p-1:]
    c.append(0)
    # C = convert_pos(c, a.n) # 正数のリストを文字列に変換する
elif c[len(c)-1] == 8:
    # 負数、繰り上がり
    c.append('9')
    # C = convert_neg(c, a.n) # 負数のリストを文字列に変換する
elif c[len(c)-1] == 9:
    # 負数、余分な9を除く
    p = len(c)
    for i in range(len(c)-1, a.n, -1):
        if c[i] == 9 and c[i-1] == 9:
            p -= 1
        else:
```

```
                break
        del c[p:]
        # C = convert_neg(c, a.n) # 負数のリストを文字列に変換する
    # オブジェクトを作成
    c.reverse()
    return RealNum(c, a.n)


def neg(a):
    if zeroP(a.val):
        return a
    A = a.val[:]
    B = get_complement(A)
    return RealNum(B, a.n)

def sub_mul(a, b):
    # a > 0 and b > 0 と仮定する
    # c = a * b を計算する
    A = a.val[:]
    B = b.val[:]
    del A[0]
    del B[0]
    A.reverse()
    B.reverse()
    N = len(A) + len(B)
    C = [0] * N
    for i, e in enumerate(B):
        t = 0
        for j, f in enumerate(A):
            g = C[i+j] + e * f + t
            t = g // 10
            C[i+j] = g % 10
        C[i+len(A)] = t
    if C[i+len(A)] != 0:
        C.append(0)
    del C[:a.n]
    C.reverse()
    return RealNum(C, a.n)

def mul(a, b):
    if a.val[0] == 0 and b.val[0] == 0: # 正数×正数
        c = sub_mul(a, b)
    elif a.val[0] == 0 and b.val[0] == 9: # 正数×負数
```

```
        d = -b
        e = sub_mul(a, d)
        c = -e
    elif a.val[0] == 9 and b.val[0] == 0: # 負数×正数
        d = -a
        e = sub_mul(d, b)
        c = -e
    elif a.val[0] == 9 and b.val[0] == 9: # 負数×負数
        d = -a
        e = -b
        c = sub_mul(d, e)
    return c

def eq(a, b):
    c = a - b
    return zeroP(c.val)

def gt(a, b):
    c = b - a
    if c.val[0] == 9:
        return True
    else:
        return False
def lt(a, b):
    c = a - b
    if c.val[0] == 9:
        return True
    else:
        return False

def copy(a):
    A = a.val[:]
    return RealNum(A, a.n)

def single(x, m, n):
    if m > n:
        s = [0, x]
        for i in range(m-1):
            s.append(0)
        return RealNum(s, n)
    elif m <= n:
        s = [0]
        for i in range(n):
```

```
            if i == n - m:
                s.append(x)
            else:
                s.append(0)
        return RealNum(s, n)

def sub_div(a, b):
    # a > 0 and b > 0 と仮定する
    # a / b を計算する
    q = []
    m = len(a.val)-1
    d = copy(a)
    for i in range(m):
        c = 1
        p = single(c, m-i, a.n)
        if b * p > d:
            q.append(0)
            continue
        while c < 10 and b * p <= d:
            c += 1
            p = single(c, m-i, a.n)
        c -= 1
        p = single(c, m-i, a.n)
        q.append(c)
        d = d - b * p
    # 必要ない 0 の範囲を計算する
    z = 0
    for i in range(m):
        if q[i] != 0:
            z = -1 # 0 を追加する必要がある
        else:
            if i < m - a.n - 1 and q[i] == 0 and q[i+1] == 0:
                continue
            elif i < m - a.n - 1 and q[i] == 0 and q[i+1] != 0:
                z = i
                break
            elif i == m - a.n - 1:
                z = i-1
                break
    if z > 0:
        del q[:z]
    elif z < 0:
        q.reverse()
```

```
        q.append(0)
        q.reverse()
    return RealNum(q, a.n)

def div(a, b):
    if a.val[0] == 0 and b.val[0] == 0: # 正数/正数
        c = sub_div(a, b)
    elif a.val[0] == 0 and b.val[0] == 9: # 正数/負数
        d = -b
        e = sub_div(a, d)
        c = -e
    elif a.val[0] == 9 and b.val[0] == 0: # 負数/正数
        d = -a
        e = sub_div(d, b)
        c = -e
    elif a.val[0] == 9 and b.val[0] == 9: # 負数/負数
        d = -a
        e = -b
        c = sub_div(d, e)
    return c

class RealNum:
    def __init__(self, val, n):
        self.val = val
        self.n = n
    __add__ = add
    __neg__ = neg
    def __sub__(a, b):
        return a + (-b)
    __mul__ = mul
    __gt__ = gt
    __lt__ = lt
    __eq__ = eq
    def __le__(a, b):
        c = b-a
        if c.val[0] == 0:
            return True
        else:
            return False
    def __ge__(self, other):
        c = self - other
        if c.val[0] == 0:
            return True
```

```
        else:
            return False
    __truediv__ = div
    def print(self):
        if self.val[0] == 0:
            val = self.val[:]
        else:
            val = get_complement(self.val)
        m = len(val)
        for i, e in enumerate(val):
            if i == 0:
                if self.val[0] == 0:
                    print(' ', end=' ')
                else:
                    print('-', end=' ')
            elif i == m-self.n:
                print('.', e, end=' ')
            else:
                print(e, end=' ')
        print()
```

を、RealNum.py という名前で保存して使っています。

実は割り算のプログラムは、商の概数を求めて、試行錯誤でやる小学校算法を使って

```
def top(a):
    for e in a.val:
        if e != 0:
            return e
    return 0
def twotop(a):
    for i, e in enumerate(a.val):
        if e != 0:
            if i < len(a.val)-2:
                return e * 10 + a.val[i+1]
            else:
                return e * 10
    return 0

def get_pos(a):
    n = 0
    while n < len(a.val) and a.val[n] == 0:
        n = n + 1
    return len(a.val)-n
```

```
def sub_div(a, b):
    # a > 0 and b > 0 と仮定する
    # a / b を整数の割り算に変更する
    ca = copy(a)
    cb = copy(b)
    if get_pos(cb) <= a.n:
        temp = strRealNum(str(10**(a.n-get_pos(cb)+1)), a.n)
        ca = ca * temp
        cb = cb * temp
##        print('temp=', end=' '); temp.print()
##    print('ca=', end= ' '); ca.print()
##    print('cb=', end= ' '); cb.print()
    m = get_pos(ca) - get_pos(cb) + a.n + 1
##    print(top(ca), top(cb), get_pos(ca), get_pos(cb), 'm=', m)
    q = []
    old_pos = get_pos(ca)
    c = top(ca) // top(cb)
    if c == 0:
        q.append(0)
    else:
        p = single(c, m, a.n)
        if cb * p <= ca:
            q.append(c)
            ca = ca - cb * p
        else:
            while cb * p > ca and c > 0:
                c -= 1
                p = single(c, m, a.n)
            p = single(c, m, a.n)
            q.append(c)
            ca = ca - cb * p
    for i in range(m-1, 0, -1):
        if get_pos(ca) == 0:
            for k in range(i):
                q.append(0)
            break
##        print('i=', i, 'q=', q)
##        print('ca=', end=' '); ca.print()
##        print('cb=', end=' '); cb.print()
##        print('## : old_pos=', old_pos, 'get_pos(ca)=', get_pos(ca))
        if get_pos(ca) < old_pos:
            c = top(ca) // top(cb)
            flag = False
```

```
        else:
            c = twotop(ca) // top(cb)
            flag = True
##        print('c=', c)
        if flag:
            old_pos = get_pos(ca) - 1
        else:
            old_pos = get_pos(ca)
        if c == 0:
            q.append(0)
        else:
            p = single(c, i, a.n)
            if cb * p <= ca:
                q.append(c)
                ca = ca - cb * p
            else:
                while cb * p > ca and c > 0:
                    c -= 1
                    p = single(c, i, a.n)
                p = single(c, i, a.n)
                q.append(c)
                ca = ca - cb * p
##    print('q=', q)
    # q の修正
    if len(q) > a.n+1:
        if q[0] > 0:
            q.reverse()
            q.append(0)
            q.reverse()
    elif len(q) == a.n+1:
        q.reverse()
        q.append(0)
        q.reverse()
    else:
        q.reverse()
        for i in range(a.n+2-len(q)):
            q.append(0)
        q.reverse()
    return RealNum(q, a.n)
def div(a, b):
    if a.val[0] == 0 and b.val[0] == 0: # 正数/正数
        c = sub_div(a, b)
    elif a.val[0] == 0 and b.val[0] == 9: # 正数/負数
```

```
        d = -b
        e = sub_div(a, d)
        c = -e
    elif a.val[0] == 9 and b.val[0] == 0: # 負数/正数
        d = -a
        e = sub_div(d, b)
        c = -e
    elif a.val[0] == 9 and b.val[0] == 9: # 負数/負数
        d = -a
        e = -b
        c = sub_div(d, e)
    return c
```

と定義しておくべきでした。これで円周率の計算が 4 倍ぐらい早くなります。

ところで、Python には、RealNum のようなクラス Decimal が定義されています。

```
from decimal import *
getcontext().prec = 1100

def marctan(n, d) :
    a = Decimal(0)
    e = Decimal(n)
    f = Decimal(0)
    e = e / d
    a = a + e
    i = 3
    while e > Decimal(0.1) ** 1100:
        e = e / d
        e = e / d
        f = e / i
        if i % 4 == 1:
            a = a + f
        else:
            a = a - f
        i = i + 2
    return a

if __name__ == '__main__':

    A = marctan(16, 5)
    B = marctan(4, 239)
    pi = A - B
    print(pi)
```

のプログラムで円周率が計算でき、一瞬で計算します。但し、

```
>>> from decimal import *
>>> getcontext().prec = 30
>>> print(Decimal(120)-Decimal(120))
0
>>> print(Decimal(12)-Decimal(120))
-108
>>> print(Decimal(1.2)-Decimal(12.0))
-10.800000000000000444089209850
>>> print(Decimal(1.1)*Decimal(1.1))
1.210000000000000019539925233403
>>> print(Decimal(12.0)-Decimal(0.120))
11.880000000000000044408920985
>>> print(Decimal(12.0)/Decimal(0.120))
100.000000000000003700743415417
>>>
```

のように、簡単な計算でも誤差が出ます。しかし、RealNum では

```
if __name__ == '__main__':
    A = strRealNum('1.2', 30)-strRealNum('12.0', 30)
    A.print()

    A = strRealNum('1.1', 30)*strRealNum('1.1', 30)
    A.print()

    A = strRealNum('12.0', 30)-strRealNum('0.120', 30)
    A.print()

    A = strRealNum('12.0', 30)/strRealNum('0.120', 30)
    A.print()
```

で

```
- 1 0 . 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  1 . 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  1 1 . 8 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  1 0 0 . 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

と、正確な値が出ます。

3.5 クラス RealNum の模範解答

ところで、Python は整数はいくらでも大きな整数の四則が正確に扱えます。この特性を使えば、RealNum は簡単に定義することが出来ます。今までの議論は何だったのだということになりますが、最後にその定義を述べておきます。

```
def zeroP(val):
    return val == 0

def set_degit(x, n):
    # 文字列をクラスにセットする数値に変換する
     if x.count('.') == 0: # 整数
        val = int(x) * 10 ** n
        return val
    else:
        while x[len(x)-1] == '0':
            x.pop()
        I = x.find('.')
        m =len(x) - I - 1
        f = int(float(x) * 10 ** m)
        val = f * 10 ** (n-m)
        return val

def strRealNum(x, n):
    val = set_degit(x, n)
    return RealNum(val, n)

def add(a, b):
    return RealNum(a.val + b.val, a.n)

def sub(a, b):
    return RealNum(a.val - b.val, a.n)

def neg(a):
    return RealNum(-a.val, a.n)

def mul(a, b):
    return RealNum((a.val* b.val) // 10**a.n, a.n)

def eq(a, b):
    return a.val == b.val

def gt(a, b):
    return a.val > b.val

def lt(a, b):
    return a.val < b.val

def copy(a):
```

```
    return RealNum(a.val, a.n)

def single(x, m, n):
    val = x * 10 ** m
    return RealNum(val, n)

def div(a, b):
    c = a.val * 10 ** a.n
    c = c // b.val
    return RealNum(c, a.n)

def intdiv(a, k):
    return RealNum(a.val // k, a.n)

class RealNum:
    def __init__(self, val, n):
        self.val = val
        self.n = n
    __add__ = add
    __neg__ = neg
    __sub__ = sub
    __mul__ = mul
    __gt__ = gt
    __lt__ = lt
    __eq__ = eq
    def __le__(a, b):
        c = b-a
        if c.val[0] == 0:
            return True
        else:
            return False
    def __ge__(self, other):
        c = self - other
        if c.val[0] == 0:
            return True
        else:
            return False
    __truediv__ = div
    __floordiv__ = intdiv
    def print(self):
        if self.val < 0:
            print('-', end='')
            v = -self.val
```

```
        else:
            v = self.val
        n = []
        while v > 0:
            n.append(v % 10)
            v //= 10
        if len(n) <= self.n:
            print('0.', end='')
            for i in range(self.n - len(n)):
                print('0', end='')
            for i in range(len(n)):
                print(int(n[len(n)-1-i]), end='')
        else:
            for i in range(len(n)):
                if i == len(n) - self.n:
                  print('.', end='')
                print(int(n[len(n)-1-i]), end='')
        print()
```

のように定義します。

```
def marctan(n, d) :
    a = strRealNum('0.0', 1100)
    e = strRealNum(str(n), 1100)
    f = strRealNum('0.0', 1100)
    e = e // d
    a = a + e
    i = 3
    l = single(1, 2, 1100)
    while e > l:
        e = e // d
        e = e // d
        f = e // i
        if i % 4 == 1:
            a = a + f
        else:
            a = a - f
        i = i + 2
    return a
if __name__ == '__main__':
    A = marctan(16, 5); print('a=', end=' '); A.print()
    B = marctan(4, 239); print('b=', end=' '); B.print()
    pi = A - B
    print('pi=', end=' '); pi.print()
```

を追加し、実行すると、直ちに

となります。

```
def marctan(n, d) :
    a = strRealNum('0.0', 1100)
    e = strRealNum(str(n), 1100)
    g = strRealNum(str(d), 1100)
    f = strRealNum('0.0', 1100)
    e = e / g
    a = a + e
    i = 3
    l = single(1, 2, 1100)
    while e > l:
```

```
        e = e / g
        e = e / g
        f = e // i
        if i % 4 == 1:
            a = a + f
        else:
            a = a - f
        i = i + 2
    return a
if __name__ == '__main__':
    A = marctan(16, 5); print('a=', end=' '); A.print()
    B = marctan(4, 239); print('b=', end=' '); B.print()
    pi = A - B
    print('pi=', end=' '); pi.print()
```

のように marctan(n, d) を定義しても、実行すると、直ちに

となります。勿論、結果は同じです。

挑戦問題

1. バイオリズムは、身体の周期が２３日、感情の周期が２８日、知性の周期が３３日で、次のように計算されます。

$$physical = \sin(T/23 * 2 * PI)$$
$$sensitivity = \sin(T/28 * 2 * PI)$$
$$intellectual = \sin(T/33 * 2 * PI)$$

ここで、T は誕生日から計算する日までの日数で、PI は円周率です。各々の値が正ならば好調、負ならば不調、零ならば好不調の変わり目の要注意日となります。ある日の前後のバイ

オリズム曲線を描く場合には、上の式で T をその日の前後に変化させて、値を線で結べばよいです。バイオリズムを表示するプログラムを作れ。