高知大学教育学部の情報数学のテキスト

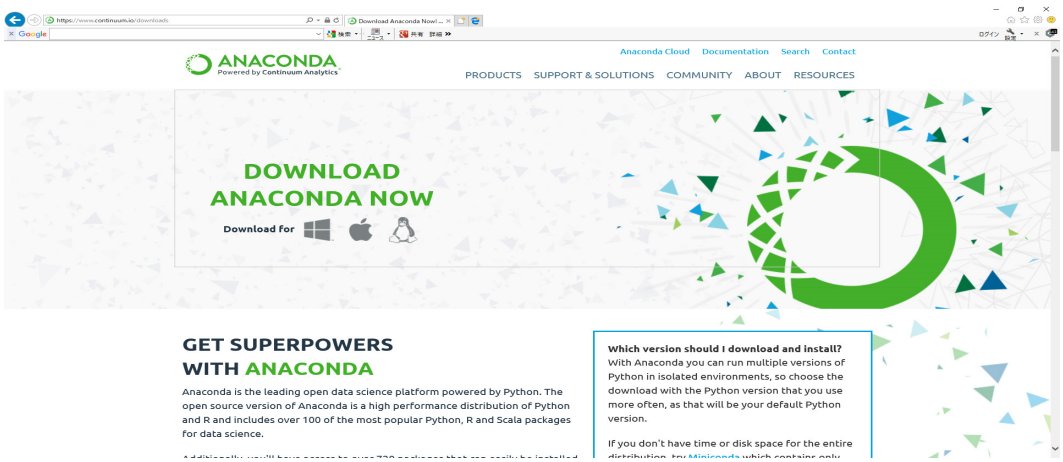
文責　：　高知大学名誉教授　中村　治

グラフィックス・アプリケーション

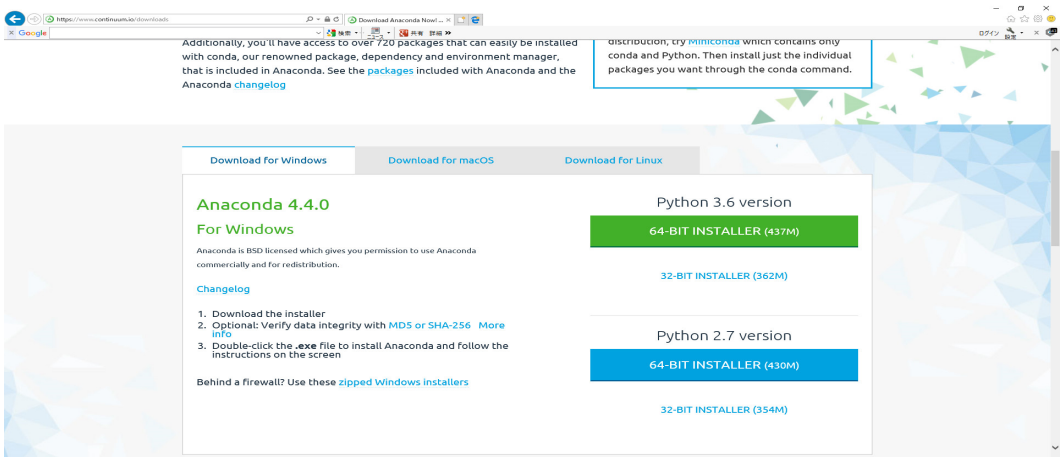
通常の Python の解説と異なり、今度は、Python の Tkinter を使って、結果が正しいかどうか見れば分かる、関数のグラフを描くことから始めます。以下のプログラミングでは、Tkinter の機能を全面的に使っているわけではなく、グラフィックス・アプリケーションに必要な機能しか使っていません。中学校・高等学校の先生になるため知っておくべき、常識（かっての高等学校の教科書「数学Ａ・数学Ｂ・数学Ｃ」のプログラミングの内容＋α）だけを述べています。

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。

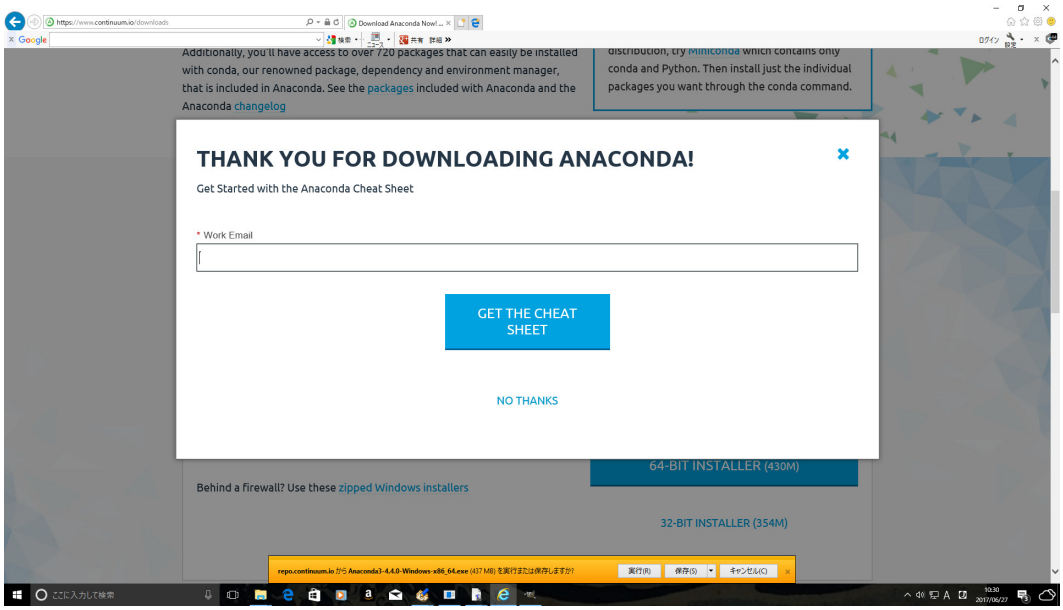


下の方の

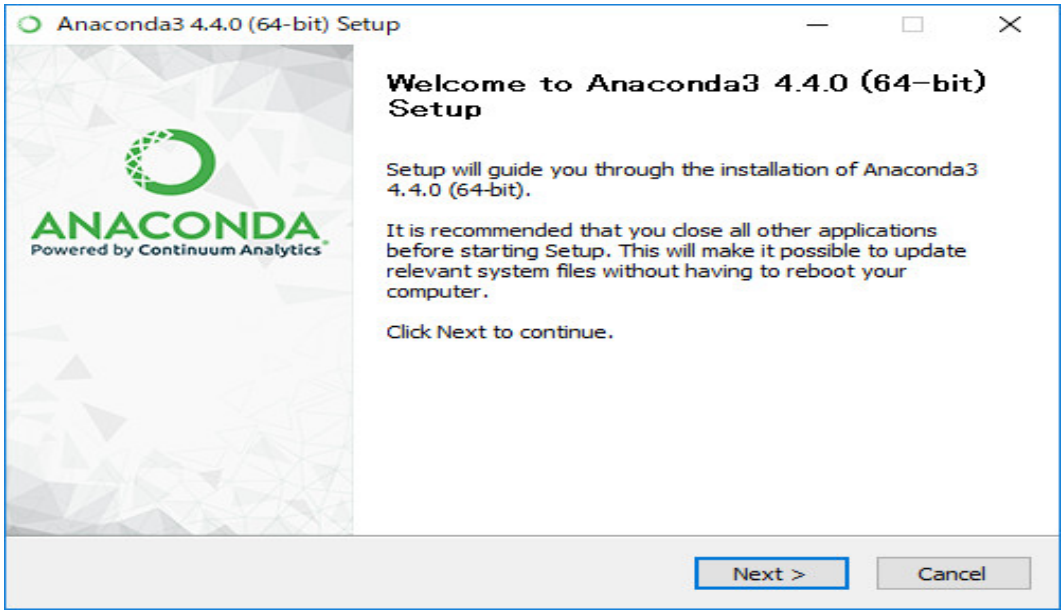


Python 3.6 version をクリックします。現在 Python2 と Python3 が利用できますが、Python 2.7 と Python 3.6 は上位互換性はなく、Python2 は新たな改善はなされないことが決まっているの

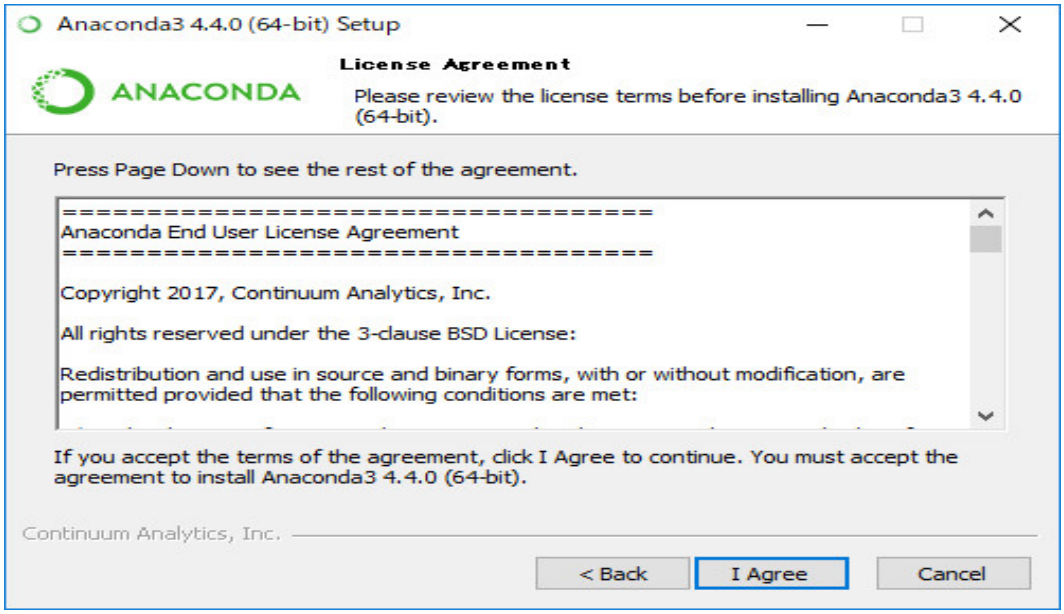
で、これからは Python3 を使って行くのが良いです。



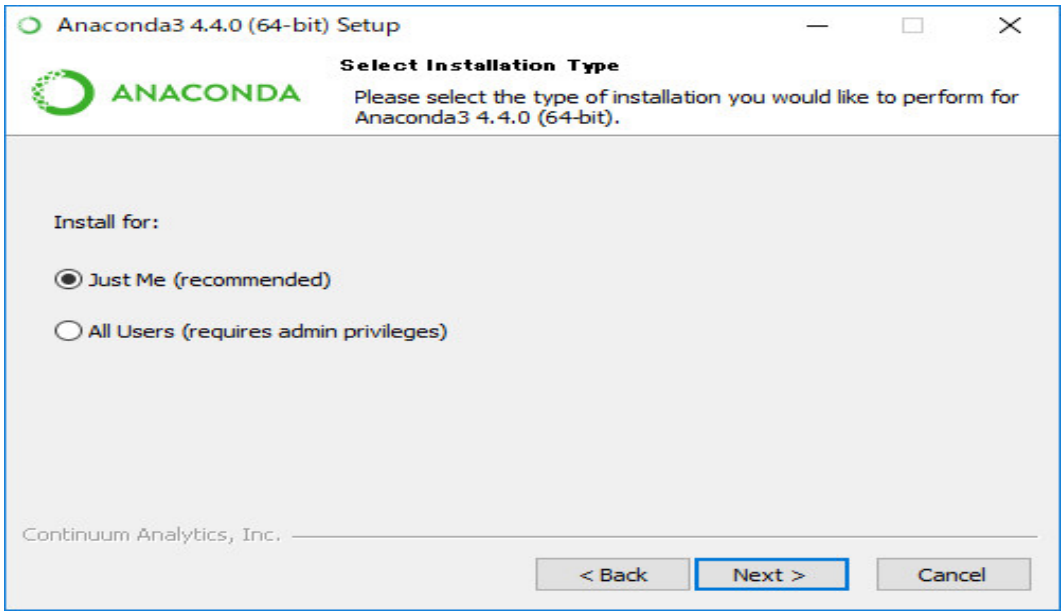
これは無視して、閉じて良いです。「実行」をクリックします。



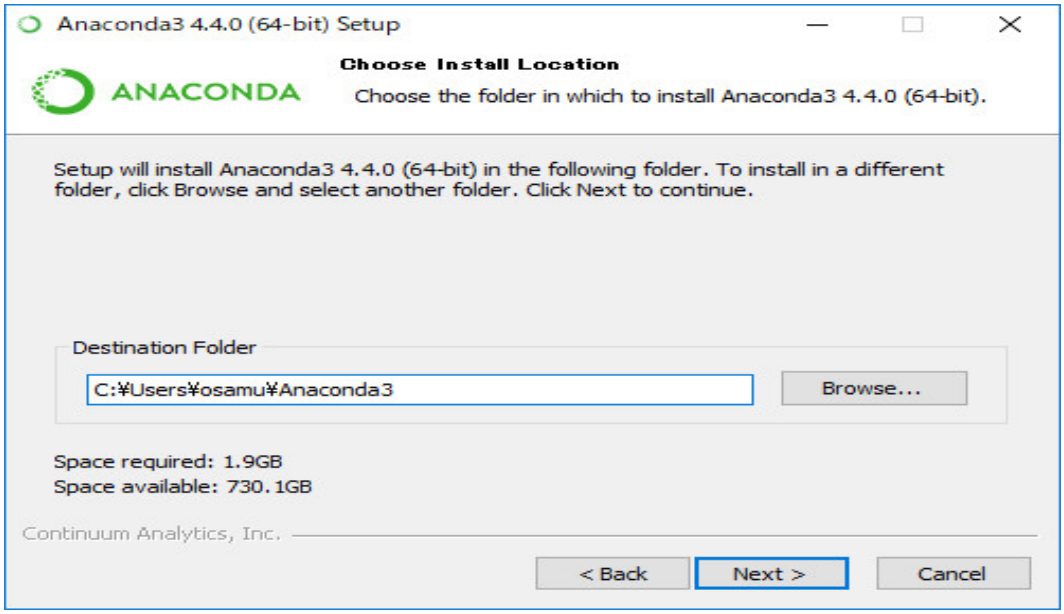
「Next」をクリックします。



「I Agree」をクリックします。



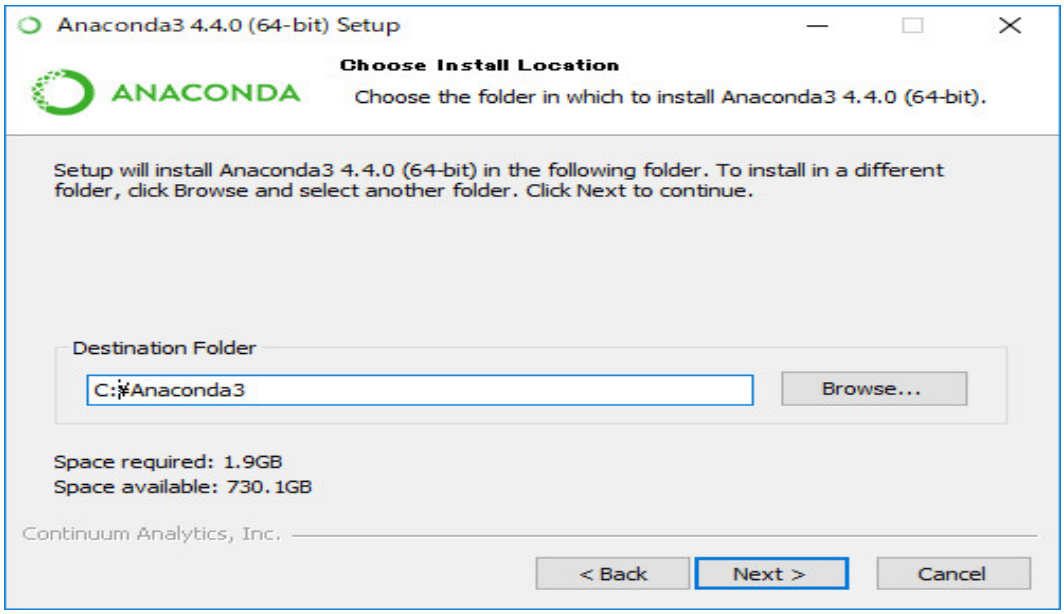
「Next」をクリックします。



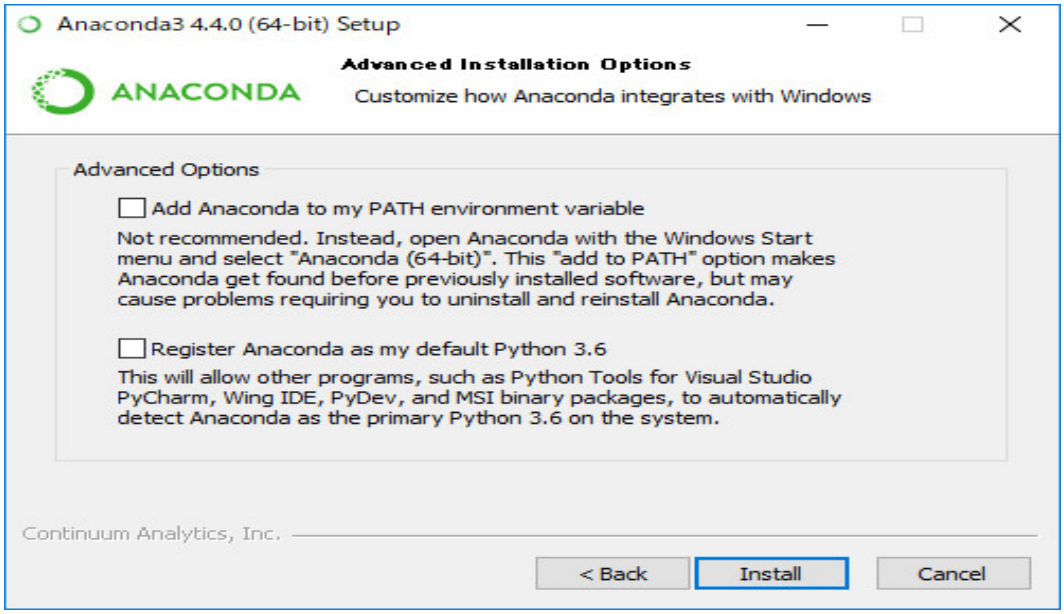
Destination Folder がデフォルトでは、後で色々問題が起こるのでここを

`C:\Anaconda3`

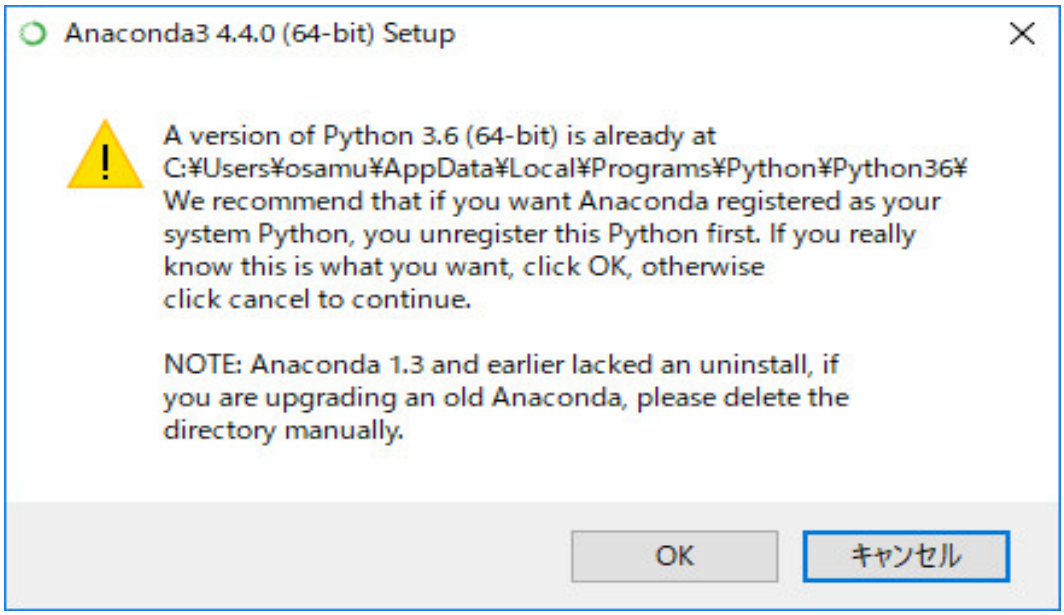
と修正します。



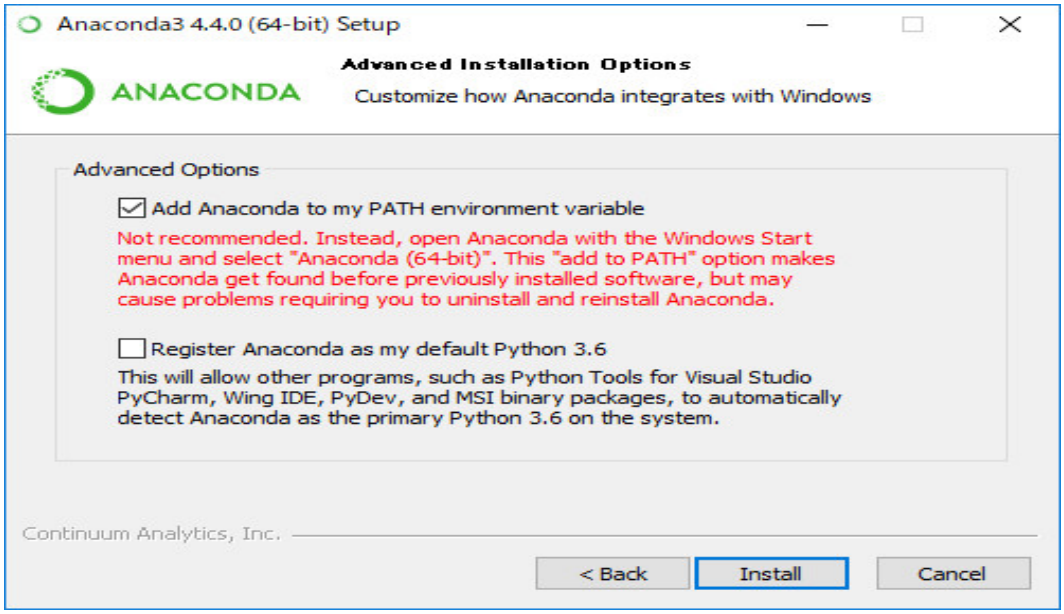
「Next」をクリックします。



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

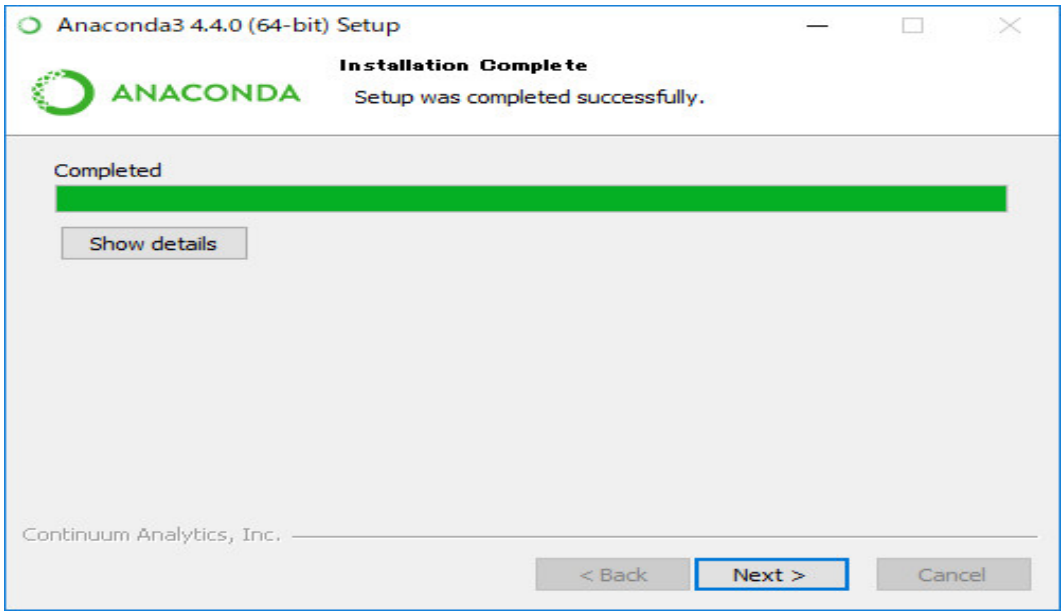


と表示されるので、「キャンセル」をクリックし、すでに Python3.6 をインストールしている場合には

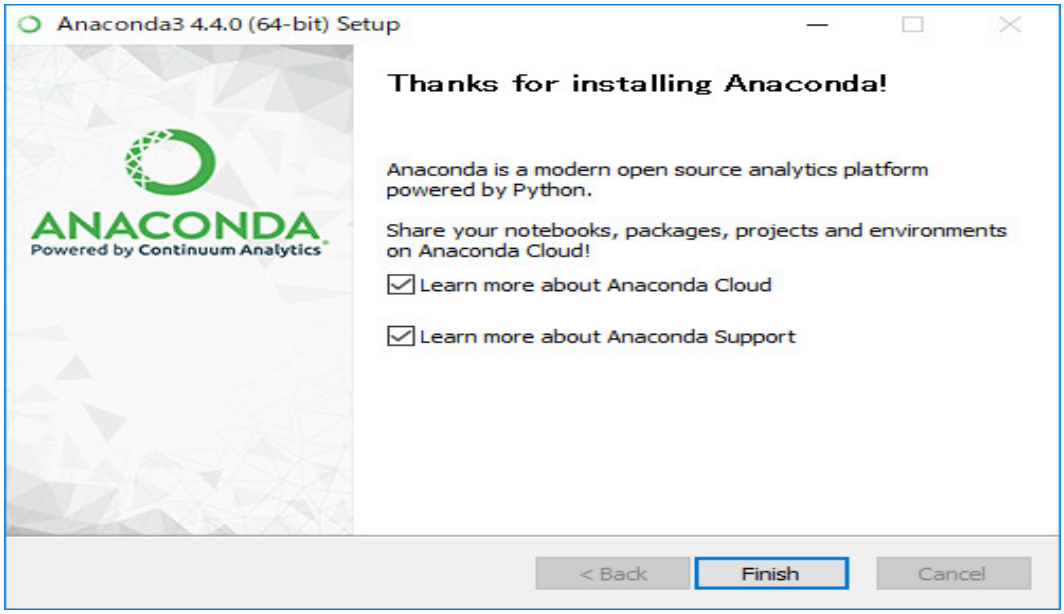


のように、上だけをチェックします。

「Install」をクリックします。

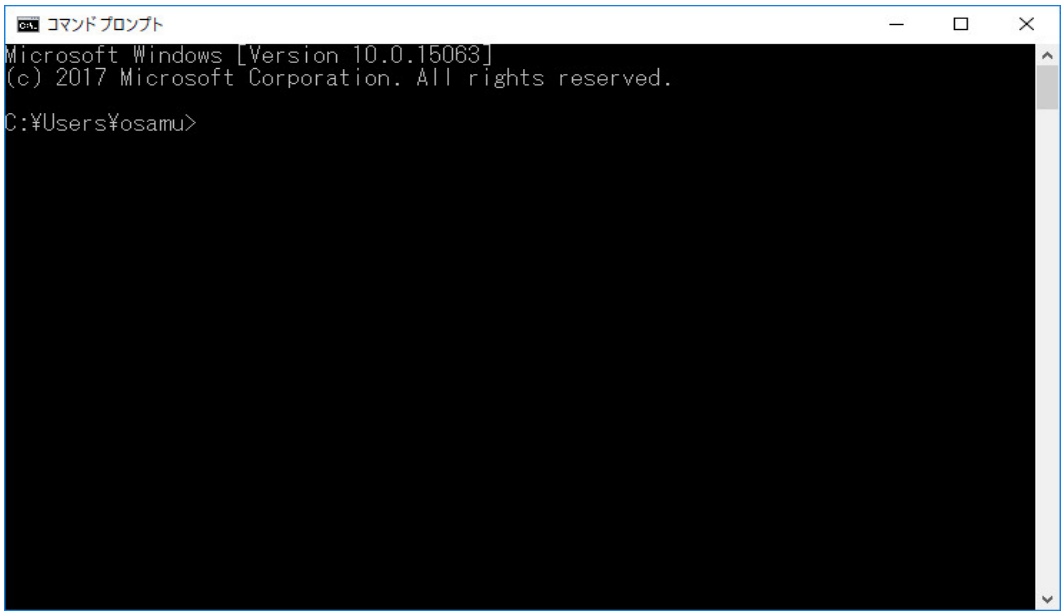


「Next」をクリックします。

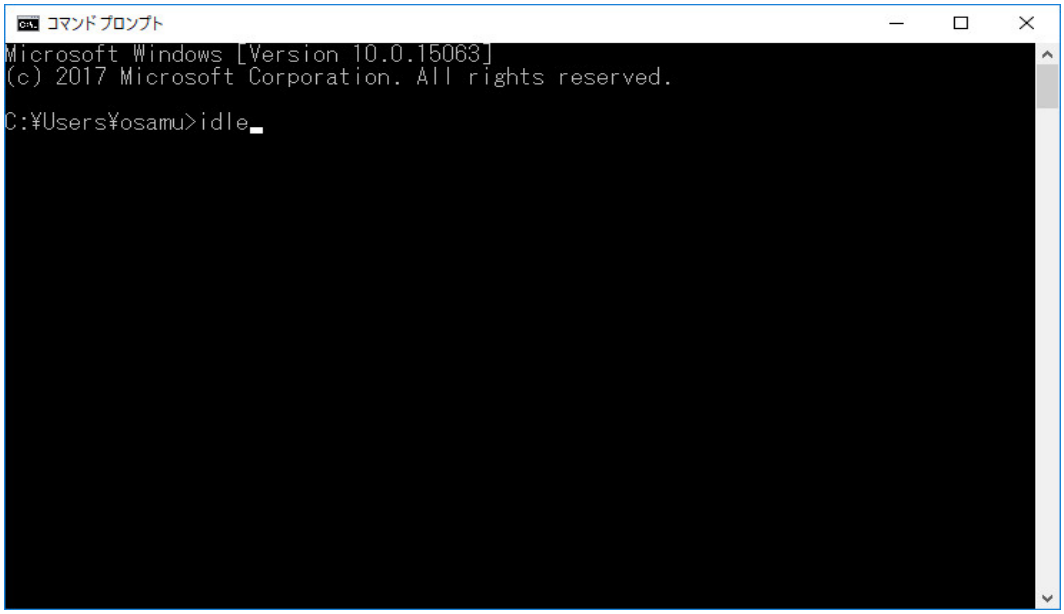


「Finish」をクリックします。

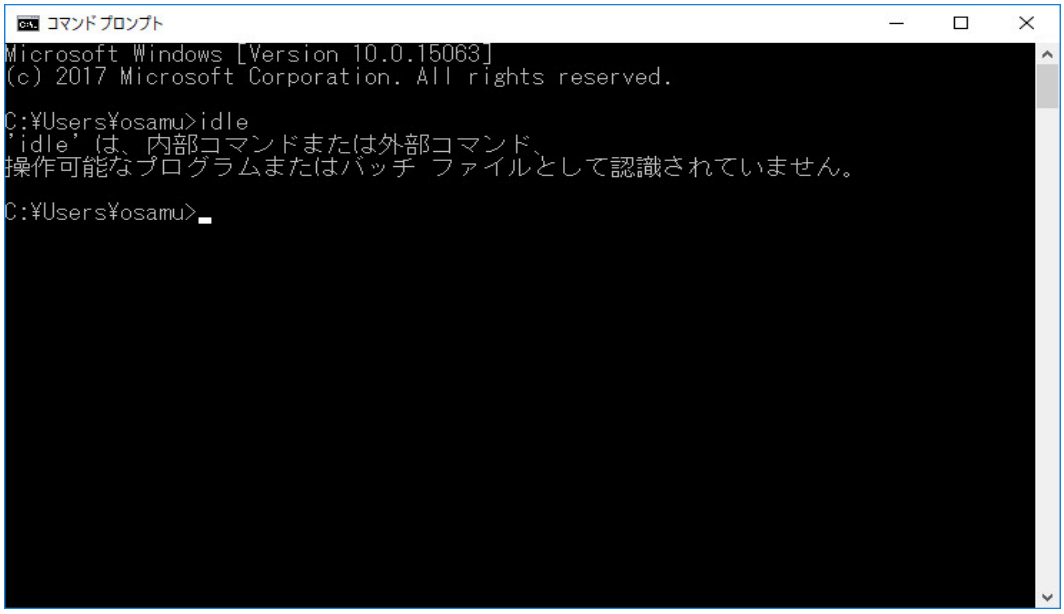
Anaconda をインストールした後に「コマンド　プロンプト」を起動し、



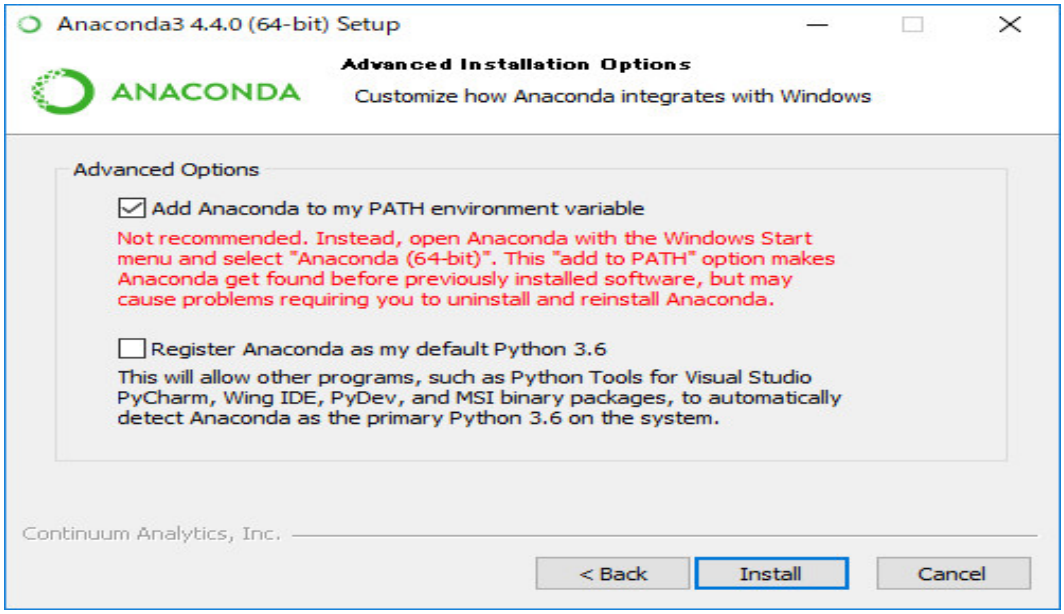
「idle」を



実行します。

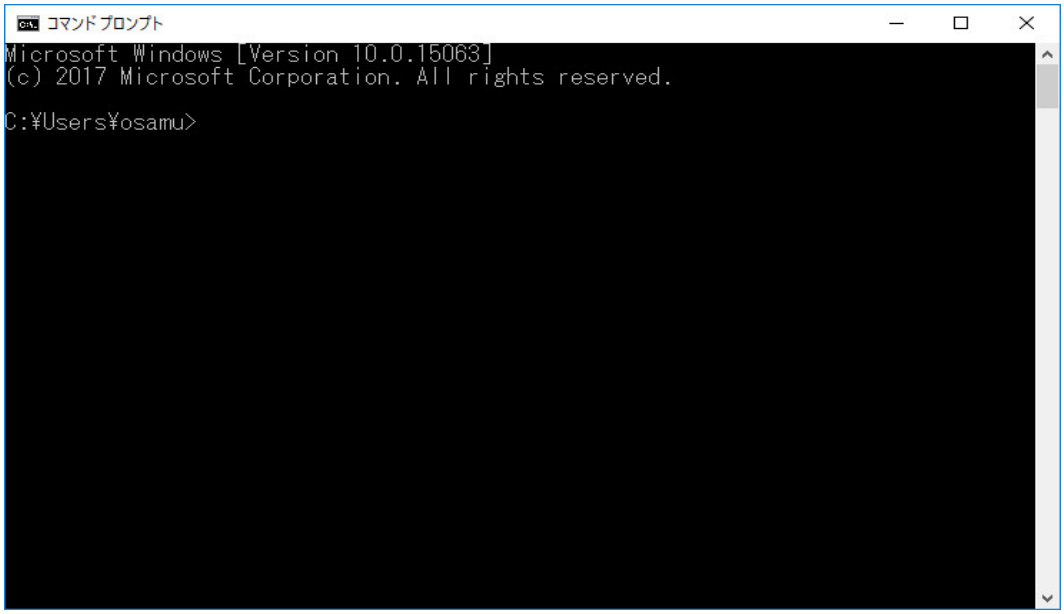


と表示されたら、

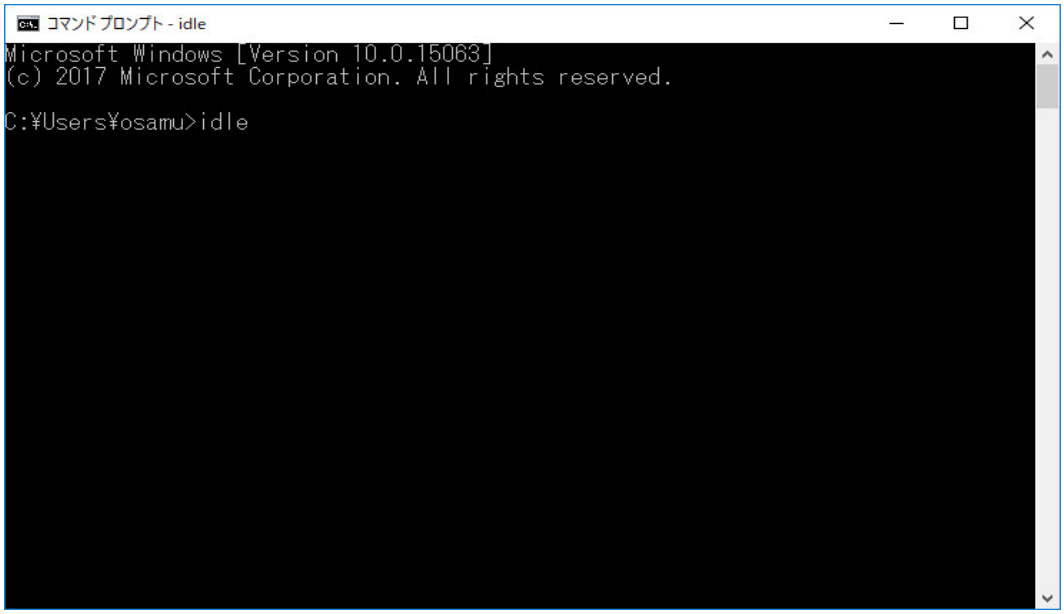


のチェックを忘れていた可能性があります。 Anaconda3 のフォルダーをエクスプローラーで削除して、もう一度インストールをやり直すのが簡単です。

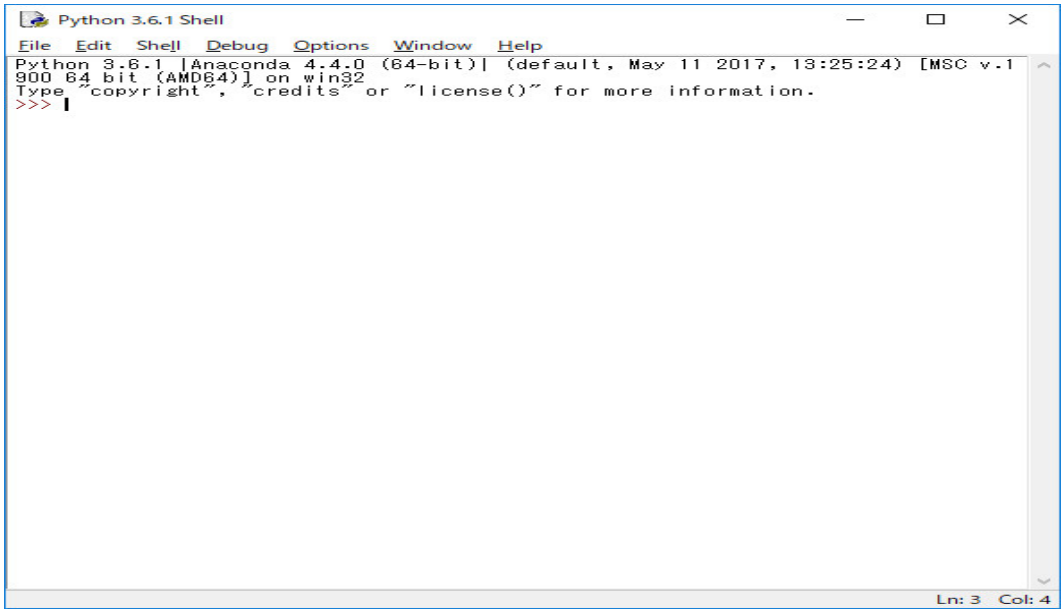
「コマンド　プロンプト」を起動し、



「idle」を

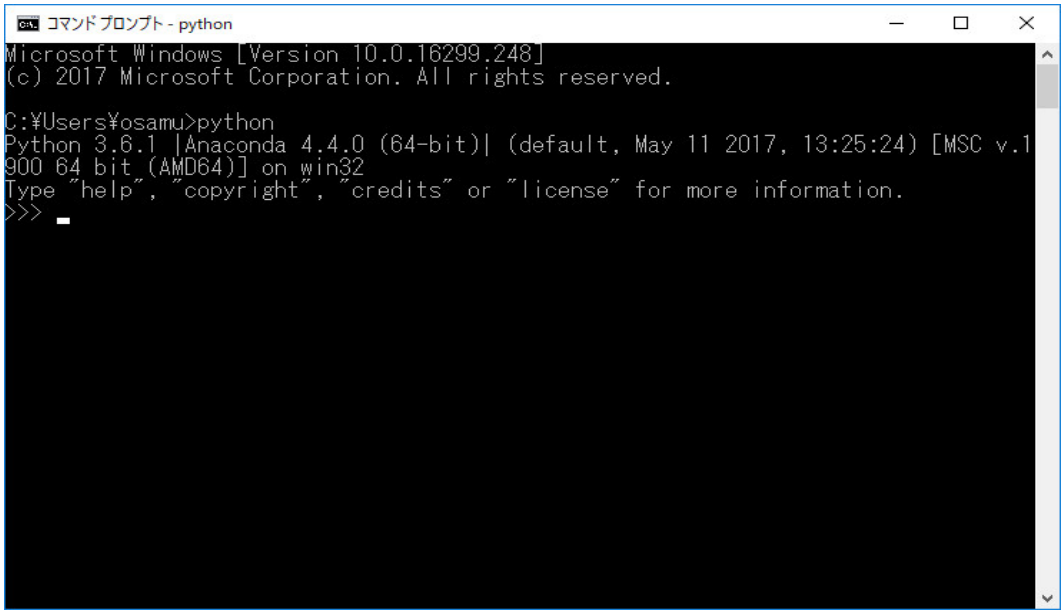


実行します。



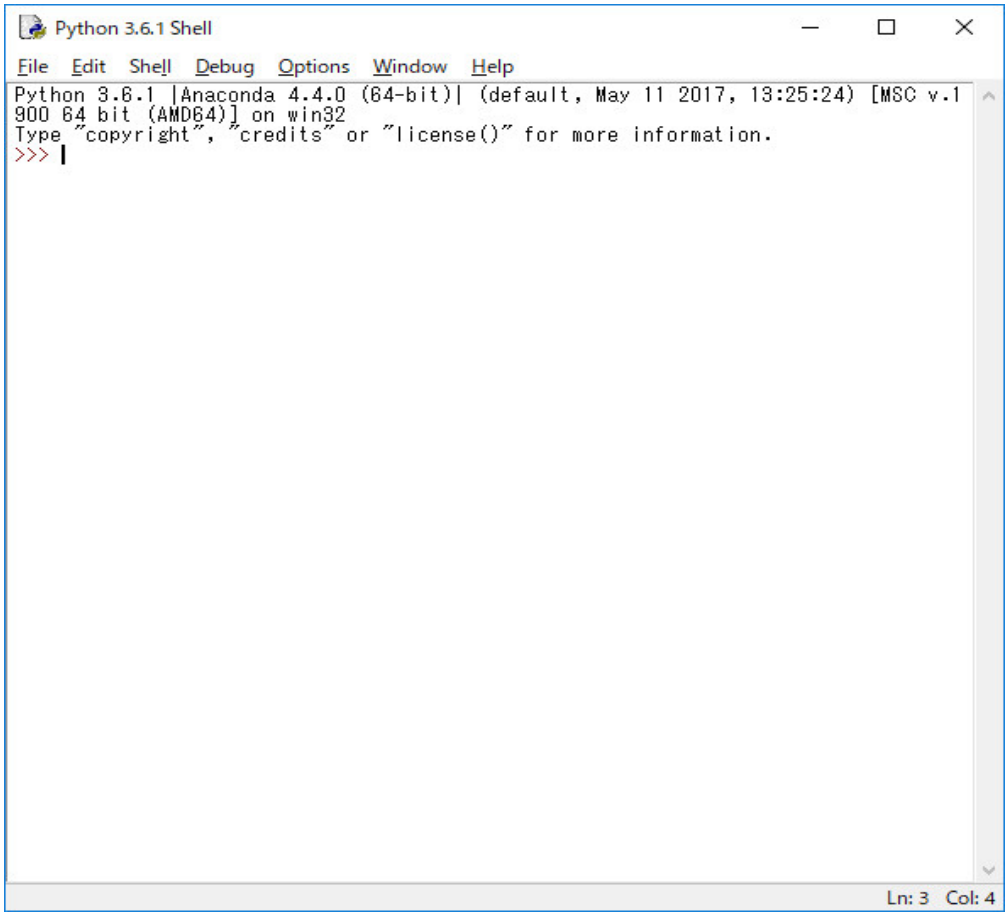
と「Python3.6.1 Shell」が起動したら、Anaconda のインストールは完成です。

「コマンド　プロンプト」を起動し、「python」を

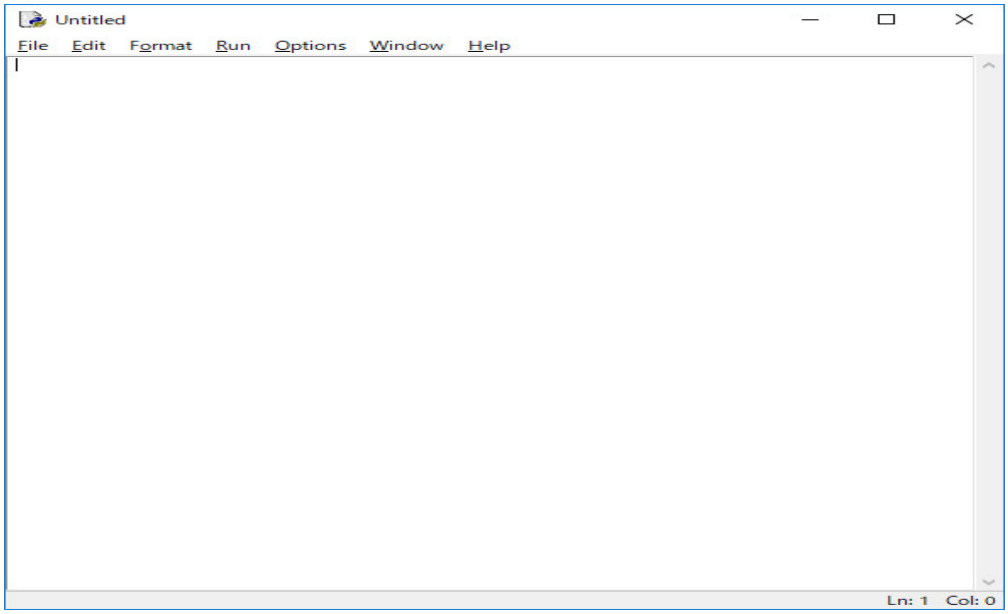


実行すると Python インタープリターが走り始めます。

それでは、Python によるグラフィックス・アプリケーションの勉強を始めます。まず Python Shell を立ち上げる。

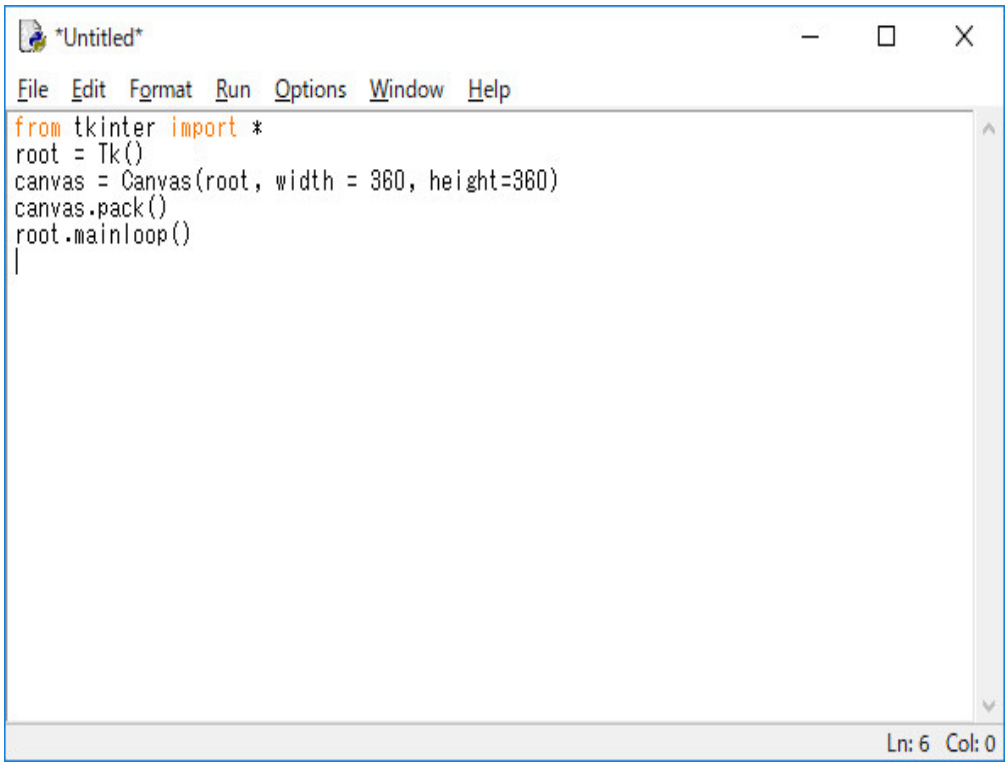


File メニューの　New File を選択する。Editor が開く。

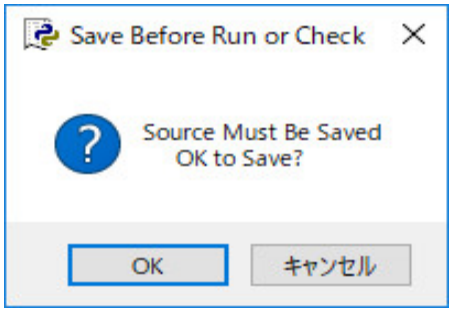


ここに、次のように打ち込む。

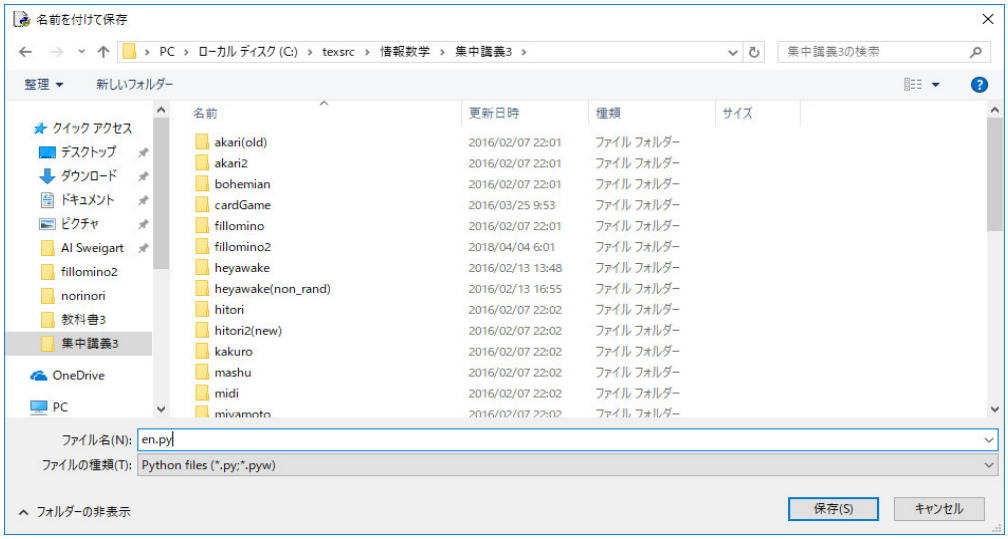
```
from tkinter import *
root = Tk()
canvas = Canvas(root, width = 360, height=360)
canvas.pack()
root.mainloop()
```



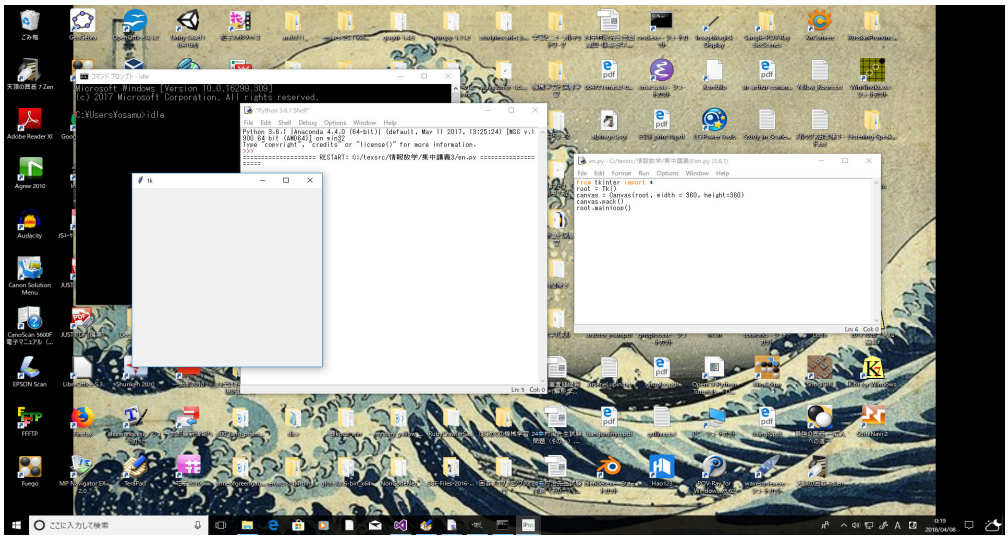
Run メニューの　Run Module を選択する。



OK のボタンをクリックする。



このプルグラムを適当なフォルダに適当な名前（例えば、en.py）を付け保存する。



360 ×　360 の空のキャンバスを持ったウインドウが表示される。

```
from tkinter import *
```

```
root = Tk()
canvas = Canvas(root, width = 360, height=360)
canvas.pack()
root.mainloop()
```

は、

```
from tkinter import *
```

で、まず tkinter を import します。tkinter は Python に最初から備わっています。次に

```
root = Tk()
```

で、top level の main window を作り、root という名前を付けています。次に、

```
canvas = Canvas(root, width = 360, height=360)
canvas.pack()
```

の 2 行で、ウィジェットを設定して window に配置します。今の場合、幅 360、高さ 360 の Canvas を canvas という名前で main window である root に設定配置しています。最後に、

```
root.mainloop()
```

で、event loop を開始して user からの要求 (event:ボタンが押されたとかマウスがクリックされたとか) を処理します。tkinter を使った window プログラミングはいつでもこのようにプログラミングを始めるものだと思って下さい。

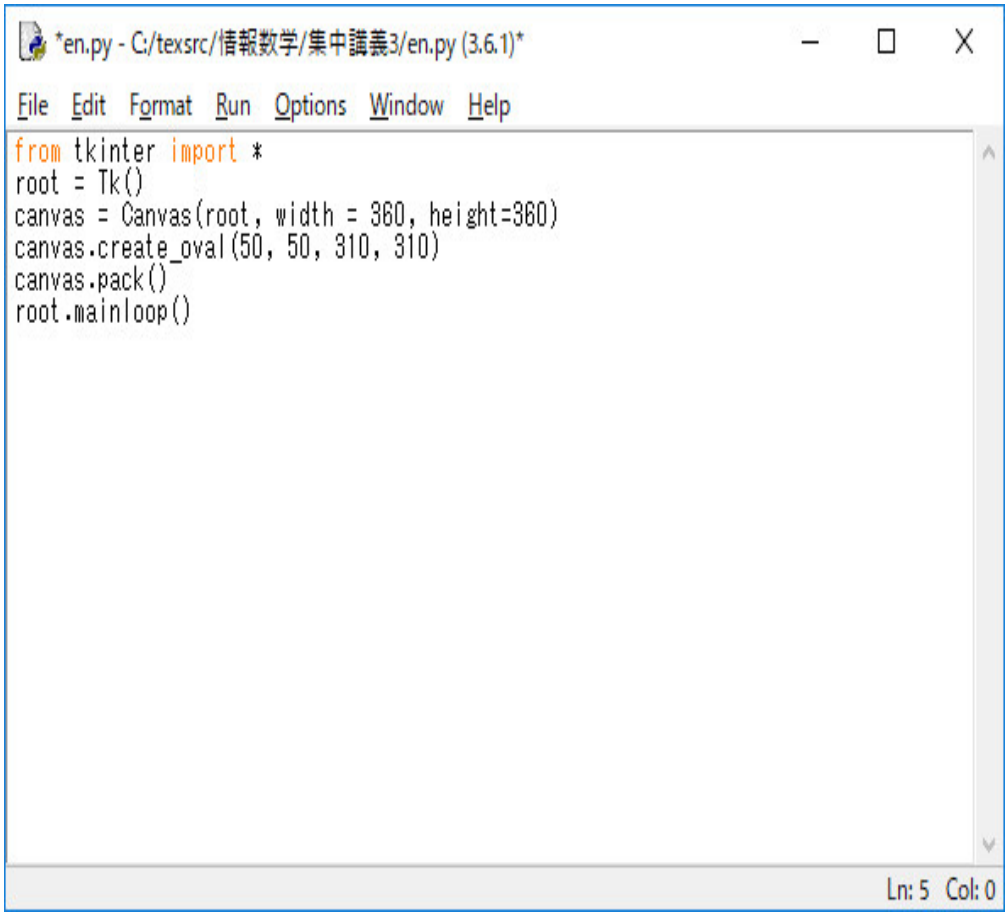
以下、このキャンバスに図形・グラフを描くことにする。図形の描画メソッドは色々あるが、まずは楕円を描くメソッドと直線を描くメソッドを使ってみる。4 行目に

```
canvas.create_oval(50, 50, 310, 310)
```

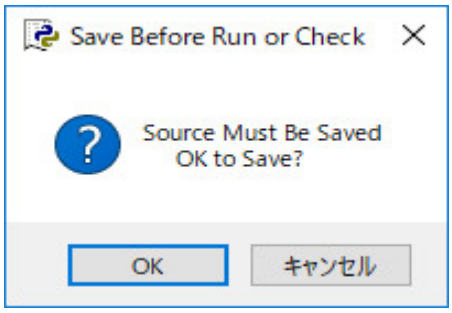
を追加する。全体のプログラムは

```
from tkinter import *
root = Tk()
canvas = Canvas(root, width = 360, height=360)
canvas.create_oval(50, 50, 310, 310)
canvas.pack()
root.mainloop()
```

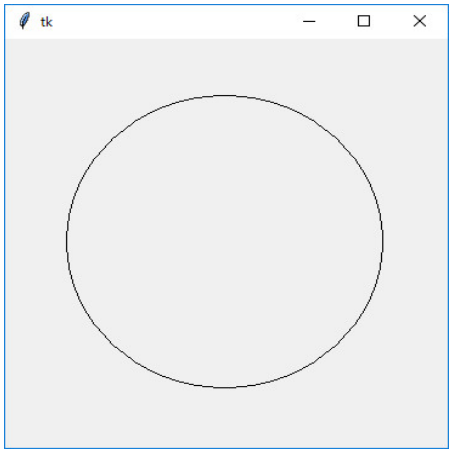
となります。



Run メニューの　Run Module を選択する。



OK のボタンをクリックする。



キャンバスに円を描きました。これは

```
canvas.create_oval(50, 50, 310, 310)
```

が、座標 (50,50) を左上端、座標 (310, 310) を右下端とする矩形（この場合は正方形）に内接する楕円（この場合は円）を描けという命令だからです。canvas の座標 (180, 180) を中心とし、半径 130 の円を描きます。キャンバスの原点が左上隅で、左右が x 座標で、上下が y 座標です。右に行けば x の値が増加し、下に行けば y の値が増加します。数学の座標系と canvas の上端の線を軸に対称になっています。

次にこの円周上の二点 $(180 + 130 * cos(t), 180 - 130 * sin(t))$ と $(180 + 130 * cos(2 * t), 180 - 130 * sin(2 * t))$ を結ぶ線分を $t = 0$ から $t = 2 * pi$ まで、$pi/80$ だけ増やしながら描きます。$180 - 130 * sin(t)$ と $-$ の符号が付いているのはキャンバスの原点が左上隅だからです。また、$cos(), sin(), pi$ という三角関数と数学の定数を使うので

```
from math import *
```

を 2 行目に書き、

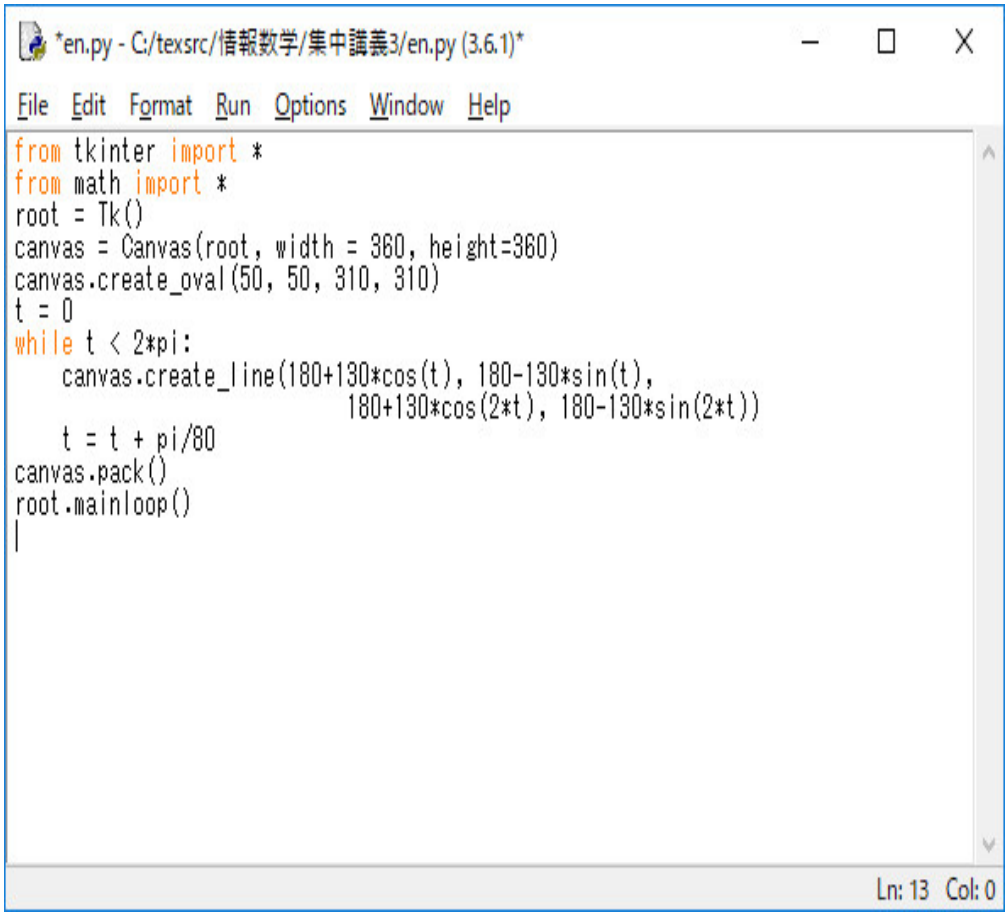
```
t = 0
while t < 2*pi:
    canvas.create_line(180+130*cos(t), 180-130*sin(t),
                       180+130*cos(2*t), 180-130*sin(2*t))
    t = t + pi/80
```

を追加し、全体のプログラムが

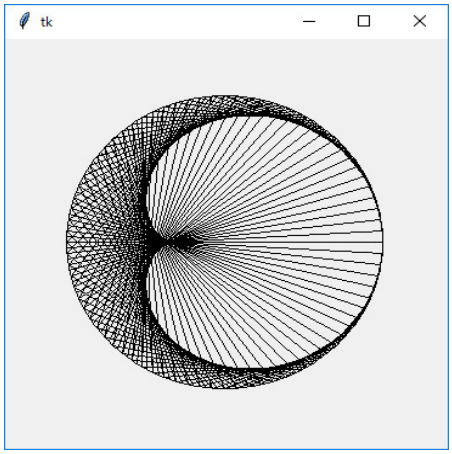
```
from tkinter import *
from math import *
root = Tk()
canvas = Canvas(root, width = 360, height=360)
canvas.create_oval(50, 50, 310, 310)
t = 0
while t < 2*pi:
    canvas.create_line(180+130*cos(t), 180-130*sin(t),
```

```
                                180+130*cos(2*t), 180-130*sin(2*t))
    t = t + pi/80
canvas.pack()
root.mainloop()
```

なるようにします。



実行すると



を描きます。

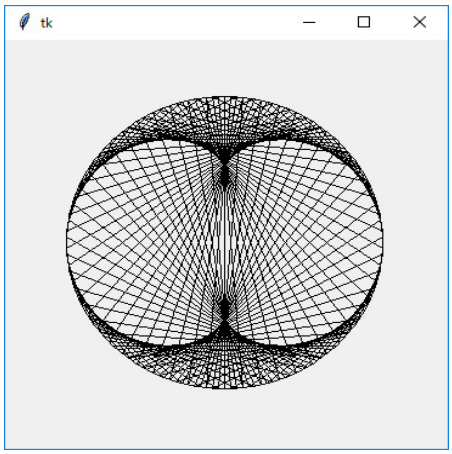
ここで

```
canvas.create_line(180+130*cos(t), 180-130*sin(t),
                           180+130*cos(2*t), 180-130*sin(2*t))
```

を

```
canvas.create_line(180+130*cos(t), 180-130*sin(t),
                           180+130*cos(3*t), 180-130*sin(3*t))
```

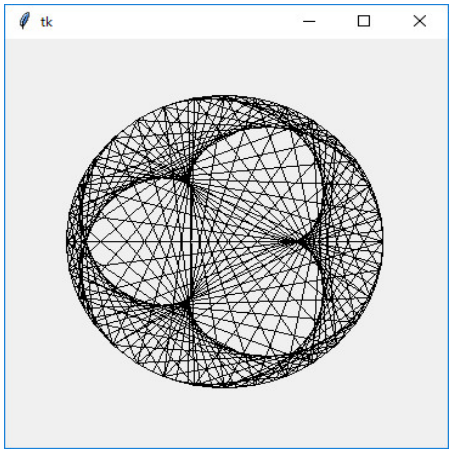
と変更すれば、



を描きます。更に

```
canvas.create_line(180+130*cos(2*t), 180-130*sin(2*t),
                           180+130*cos(5*t), 180-130*sin(5*t))
```

と変更すれば、

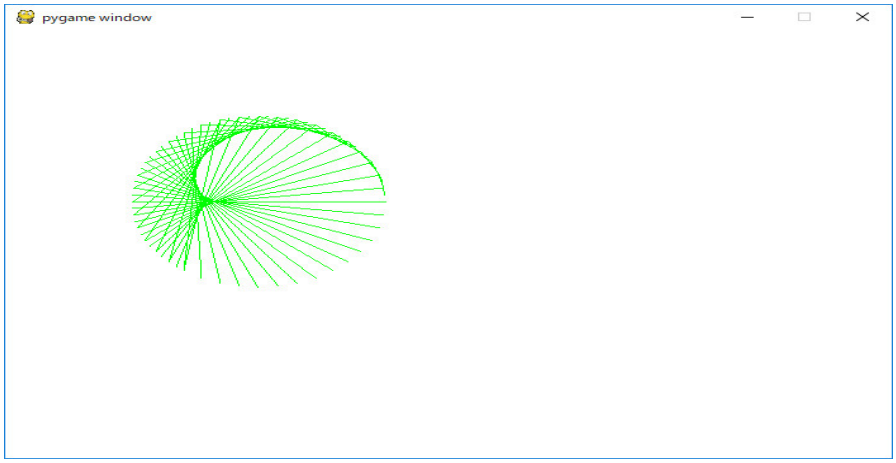


を描きます。色々思いついたことをやってみるのが勉強になります。自分で見付けたことは宝物になります。

Python というプログラミング言語と Pygame というパッケジを使うと簡単にアニメーションが出来ます。Pygame は Python をインストールしても自動的にインストールされませんので、インターネットで調べてインストールしてください。簡単に出来ます。**但し、Anaconda を使って Python をインストールしていれば、自動的にインストールされています。**

```
import pygame
from math import *

pygame.init()

black = (0,0,0)
white = (255, 255, 255)
green = (0, 255, 0)
red= (255, 0, 0)

size = (700, 500)

screen = pygame.display.set_mode(size)

pi = atan(1.0)*4
done = False
clock = pygame.time.Clock()
t = pi
while done == False:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True
    screen.fill(white)
    x = 0
    while x <= 2*pi-2*t:
        pygame.draw.line(screen, green, [200+100*cos(x), 200-100*sin(x)],\
                         [200+100*cos(2*x), 200-100*sin(2*x)], 1)
        x += pi/40
    t -= pi/40
    pygame.display.flip()
    if t < 0:
        t = pi
    clock.tick(1)

pygame.quit()
```

実行すると次のようなアニメーションが始まります。



この講義では時間がありませんから、Pygame の入門講座はしませんし、このプログラムの解説もしませんが、興味があれば、

`http://www.youtube.com/playlistlist=PL1D91F4E6E79E73E1?`

を見て下さい。 Python と Pygame の基本の基本と簡単なアーケードゲームの作り方が youtube で 15 時間ぐらいで説明されています。英語ですが、見ていれば何をしているか分かるはずです、また、

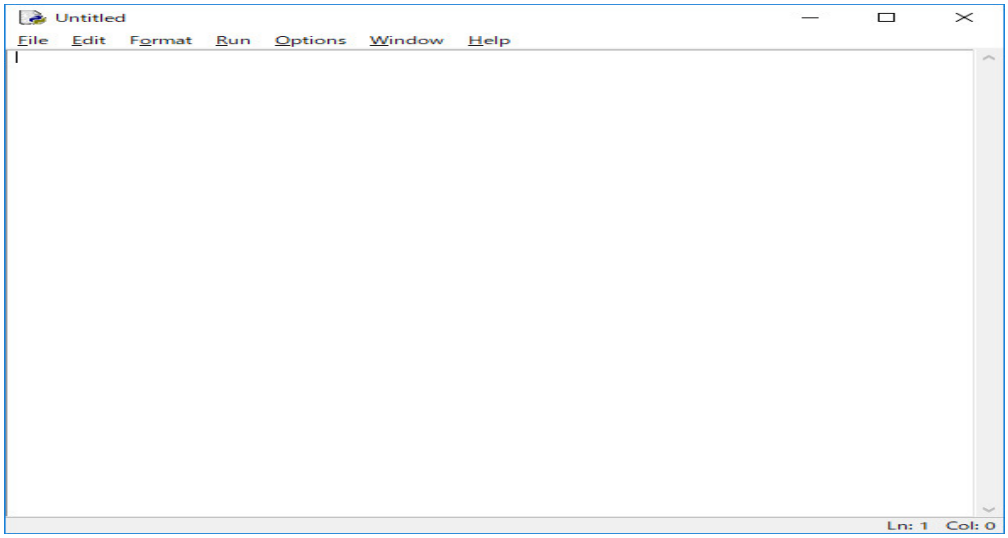
`http://ProgramArcadeGame.com`

には on-line のテキストがあります。アメリカの Simpson College の Dr. Paul Vincent Craven 氏の講義録です。更に、Linux でのものですが、Pygame や Python の説明が youtube に沢山あります。

勿論、このようなプログラムを Timer を使って Visual C++ や C++ Builder で作れますが、この講義のレベルを超えています。複雑なアニメーションやゲームを作るには Visual C++ や C++ Builder で作りますが、このような単純なものは、イギリスで開発された子供の教育用の手のひらに乗るコンピュータ Raspberry Pi の言語としても採用されている Python で作る方が簡単です。Python はオブジェクト指向言語です。更に、Scratch という子供用のプログラミング言語でアニメーションやゲームを作ることもできます。

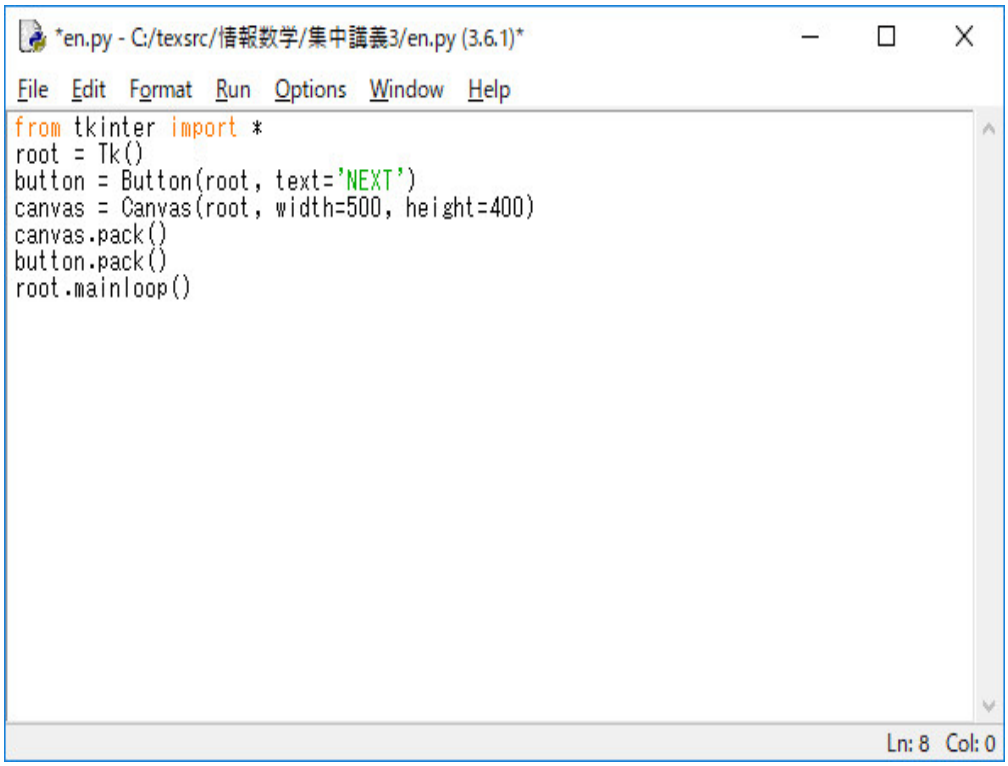
別のプログラム（関数 $y = exp(x)$ を表示する）を作ってみます。

File メニューの　New File を選択する。Editor が開く。



ここに、次のように打ち込む。

```
from tkinter import *
root = Tk()
button = Button(root, text='NEXT')
canvas = Canvas(root, width=500, height=400)
canvas.pack()
button.pack()
root.mainloop()
```



違いは

```
button = Button(root, text='NEXT')
```

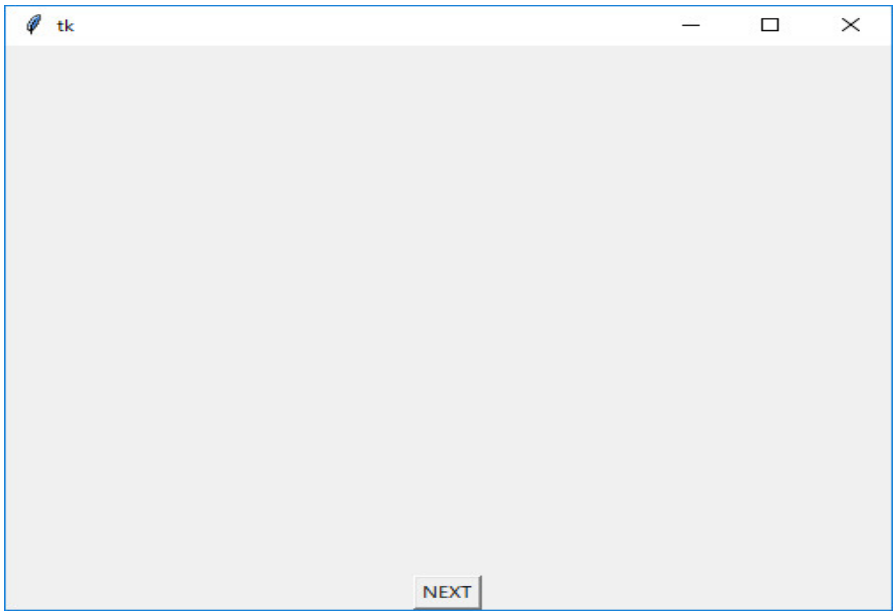
と

```
button.pack()
```

が追加されたことと Canvas のサイズが変更されていることです。

```
button = Button(root, text='NEXT')
button.pack()
```

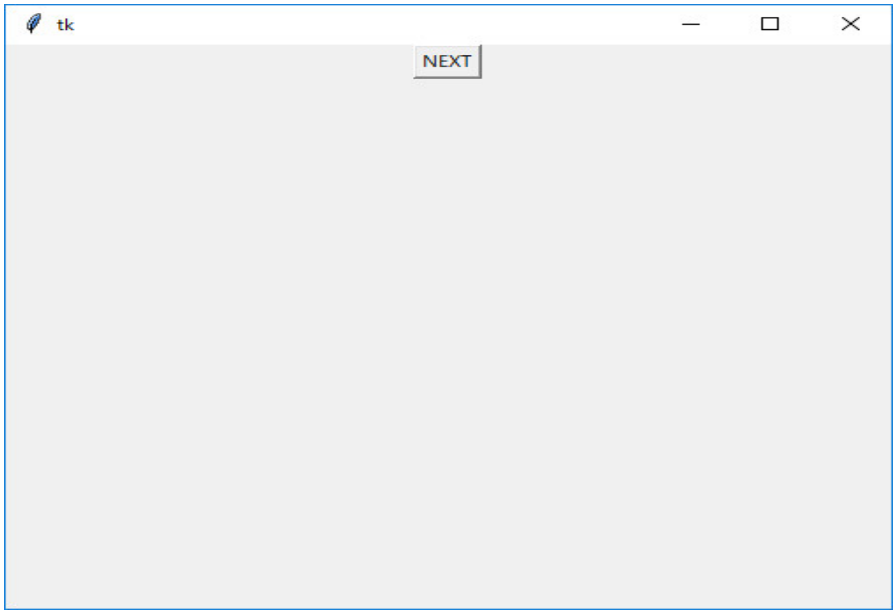
は、NEXT と表示される Button を root に設定配置します。Run メニューの　Run Module を選択し、OK のボタンをクリックし、このプルグラムを適当なフォルダに適当な名前（例えば、exp.py）を付け保存する。



となります。ここで順番を変えて

```
from tkinter import *
root = Tk()
button = Button(root, text='NEXT')
canvas = Canvas(root, width=500, height=400)
button.pack()
canvas.pack()
root.mainloop()
```

とすると



となります。どちらでも好きな方で良いですがここでは元の方を使います。つぎに

```
button = Button(root, text='NEXT')
```

を

```
button = Button(root, text='NEXT', command=paint)
```

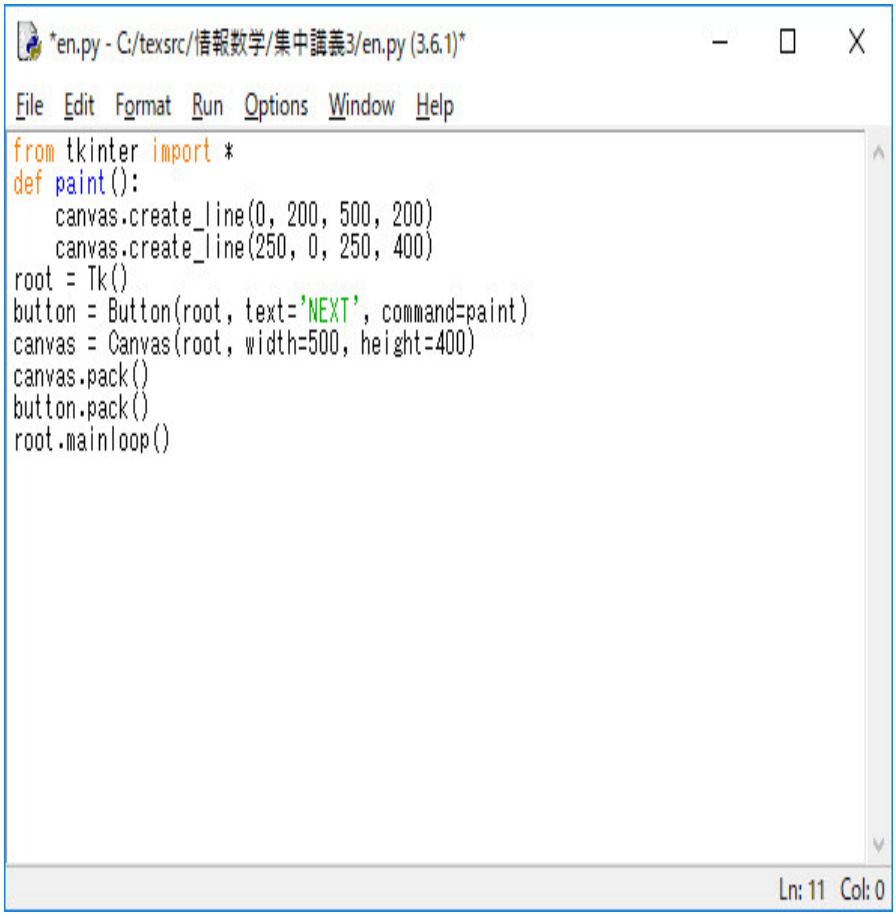
に変更する。これは、ボタンをクリックしたとき実行する関数名（今の場合、paint）を指定しています。paint の定義を書きます。

```
def paint():
    canvas.create_line(0, 200, 500, 200)
    canvas.create_line(250, 0, 250, 400)
```

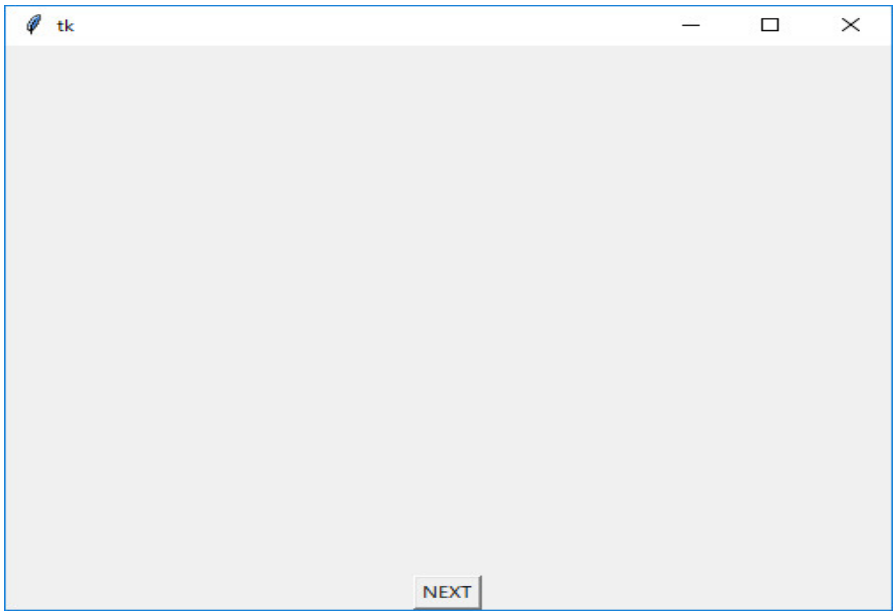
を先頭に追加し、全体のプログラムは

```
from tkinter import *
def paint():
    canvas.create_line(0, 200, 500, 200)
    canvas.create_line(250, 0, 250, 400)
root = Tk()
button = Button(root, text='NEXT', command=paint)
canvas = Canvas(root, width=500, height=400)
canvas.pack()
button.pack()
root.mainloop()
```

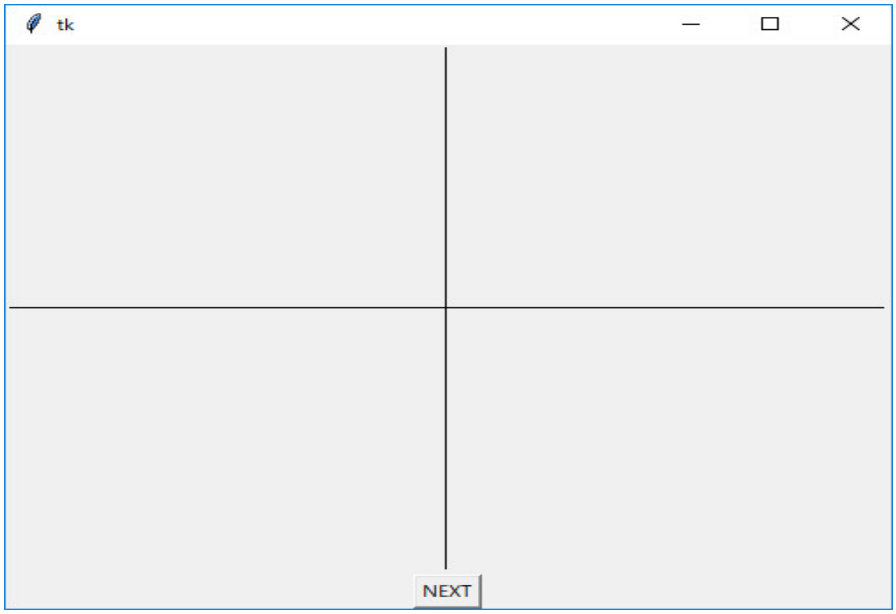
となります。



Run メニューの　Run Module を選択し、OK のボタンをクリックし、プログラムを実行します。



NEXT のボタンをクリックします。



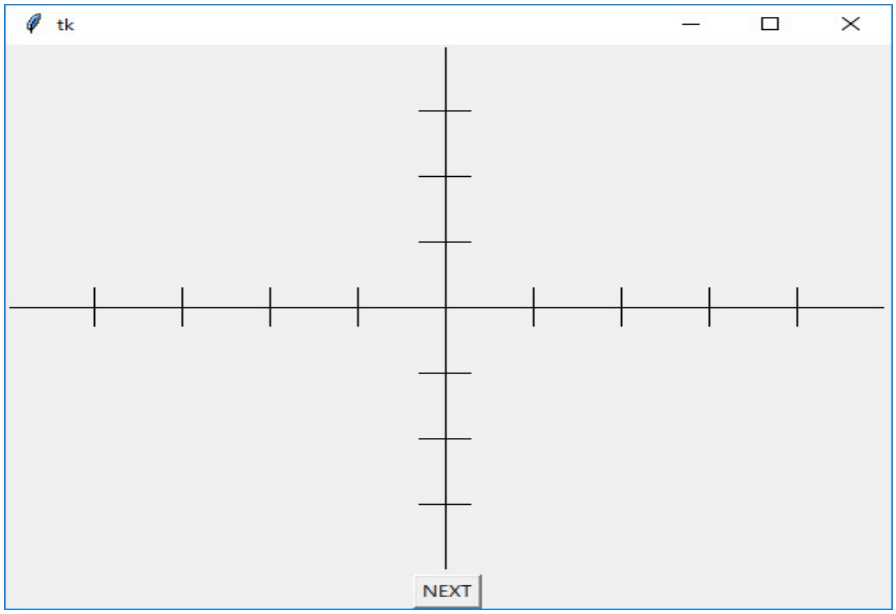
座標軸を表示します。次に、paint の定義を

```
def paint():
    canvas.create_line(0, 200, 500, 200)
    canvas.create_line(250, 0, 250, 400)
    for x in range(0, 500, 50):
        canvas.create_line(x, 185, x, 215)
    for y in range(0, 400, 50):
        canvas.create_line(235, y, 265, y)
```

と修正し、全体のプログラムは

```
from tkinter import *
def paint():
    canvas.create_line(0, 200, 500, 200)
    canvas.create_line(250, 0, 250, 400)
    for x in range(0, 500, 50):
        canvas.create_line(x, 185, x, 215)
    for y in range(0, 400, 50):
        canvas.create_line(235, y, 265, y)
root = Tk()
button = Button(root, text='NEXT', command=paint)
canvas = Canvas(root, width=500, height=400)
canvas.pack()
button.pack()
root.mainloop()
```

とします。実行し、NEXT ボタンをクリックすると



と目盛りを表示します。次に、関数 $y = exp(x)$ $(5 <= x <= 5)$ を描きます。paint の定義を

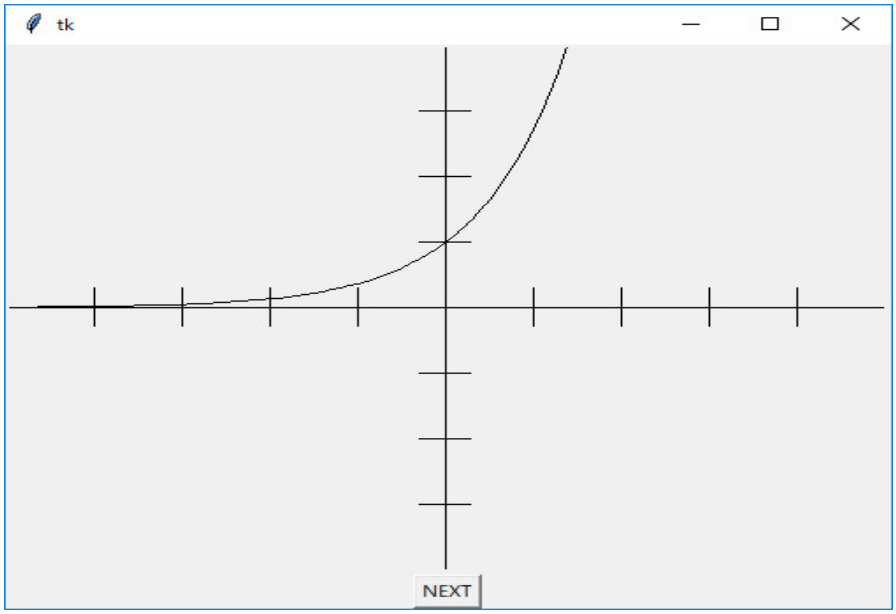
```
def paint():
    canvas.create_line(0, 200, 500, 200)
    canvas.create_line(250, 0, 250, 400)
    for x in range(0, 500, 50):
        canvas.create_line(x, 185, x, 215)
    for y in range(0, 400, 50):
        canvas.create_line(235, y, 265, y)
    x = -5
    dx = 0.1
    while x <= 5:
        canvas.create_line(250+50*x, 200-50*exp(x),
                           250+50*(x+dx), 200-50*exp(x+dx))
        x = x + dx
```

と修正し、全体のプログラムは

```
from tkinter import *
from math import *
def paint():
    canvas.create_line(0, 200, 500, 200)
    canvas.create_line(250, 0, 250, 400)
    for x in range(0, 500, 50):
        canvas.create_line(x, 185, x, 215)
    for y in range(0, 400, 50):
        canvas.create_line(235, y, 265, y)
    x = -5
```

```
    dx = 0.1
    while x <= 5:
        canvas.create_line(250+50*x, 200-50*exp(x),
                           250+50*(x+dx), 200-50*exp(x+dx))
        x = x + dx
root = Tk()
button = Button(root, text='NEXT', command=paint)
canvas = Canvas(root, width=500, height=400)
canvas.pack()
button.pack()
root.mainloop()
```

とします。実行し、NEXT ボタンをクリックすると



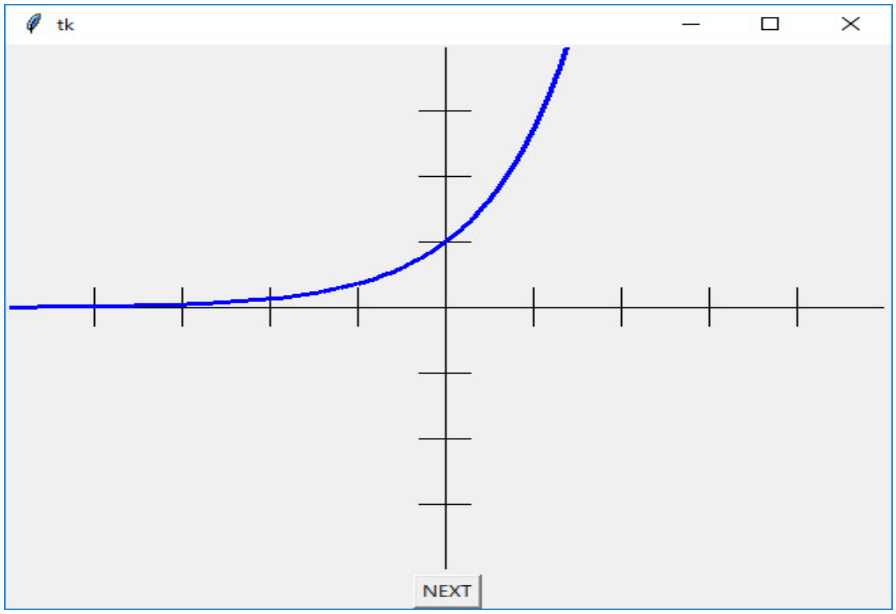
となります。座標軸と $y = exp(x)$ の関数が `-5 <= x <= 5` の範囲で表示されている。

```
        canvas.create_line(250+50*x, 200-50*exp(x),
                           250+50*(x+dx), 200-50*exp(x+dx))
```

を

```
        canvas.create_line(250+50*x, 200-50*exp(x),
                           250+50*(x+dx), 200-50*exp(x+dx),
                           fill='blue', width=3.0)
```

と修正します。実行し、NEXT ボタンをクリックすると



と関数 $y = exp(x)$ 5 <= x <= 5 を blue で線の幅 3 ドットで描きます。

このプログラムを作り替えて、Next のボタンを押すたびに、exp(x) の級数展開の近似式

$$e^x \sim 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} = \sum_{i=0}^{n} \frac{x^i}{i!}$$

を順次表示するようにする。

まず、階乗を計算する関数 $fact(n)$ と最初の $n+1$ 項までの和を計算する関数 $expN(n, x)$ を作る。

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

これを

```
def paint():
```

の前に打ち込む。更に、その前に gloval 変数 $N = -1$ を定義しておく。

```
N = -1
```

そして、Next のボタンを押すたびに N を -1 から順に $0, 1, 2, 3, 4,$ と増やしながら $n = N$ として $expN(n, x)$ のグラフを表示すれば良い。paint() を次のように修正する。

```
def paint():
    global N
    n = N
    canvas.create_line(0, 200, 500, 200)
    canvas.create_line(250, 0, 250, 400)
    for x in range(0, 500, 50):
        canvas.create_line(x, 185, x, 215)
    for y in range(0, 400, 50):
        canvas.create_line(235, y, 265, y)
    x = -5
    dx = 0.1
    while x <= 5:
        canvas.create_line(250+50*x, 200-50*exp(x),
                           250+50*(x+dx), 200-50*exp(x+dx),
                           fill='blue', width=3.0)
        x = x + dx
    if n >= 0:
        x = -5
        dx = 0.1
        while x <= 5:
            canvas.create_line(250+50*x, 200-50*expN(n, x),
                               250+50*(x+dx), 200-50*expN(n, x+dx),
                               fill='red', width=3.0)
            x = x + dx
    N = N+1
```

先頭にある

```
    global N
```

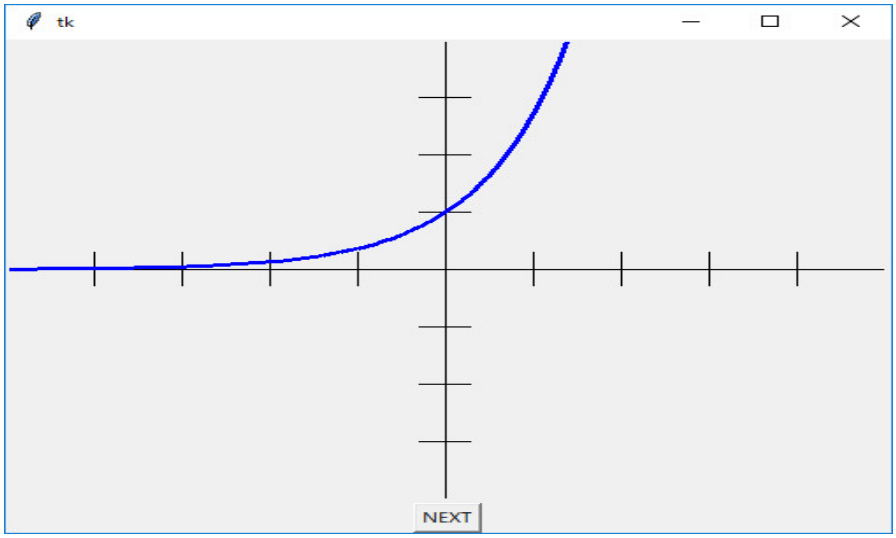
で、N はグローバル変数であることを宣言している。グローバル変数とは、関数内だけで定義された変数（関数内だけで有効な変数：ローカル変数という）ではなく、関数の外で定義された変数で、値を変更すれば、関数の実行が終わった後もその変更が有効であるということです。値を変更しなければ、グローバル変数であることを宣言しなくてもいいです。

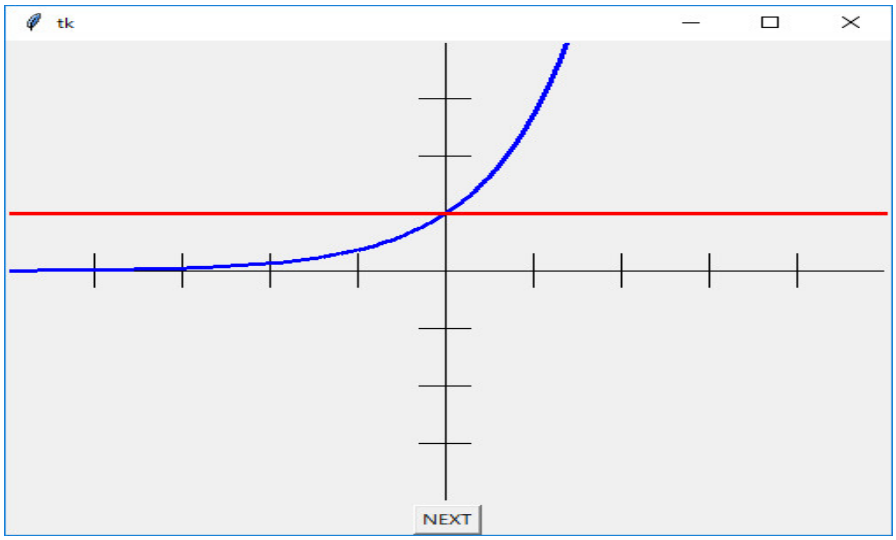
全体のプログラムは

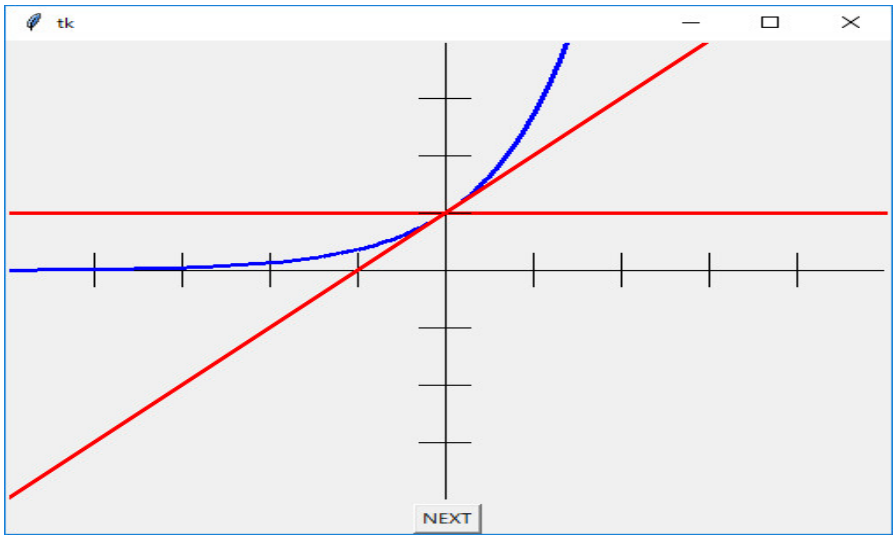
```
from tkinter import *
from math import *
N = -1
def fact(n):
    r = 1.0
    for i in range(1, n+1, 1):
```

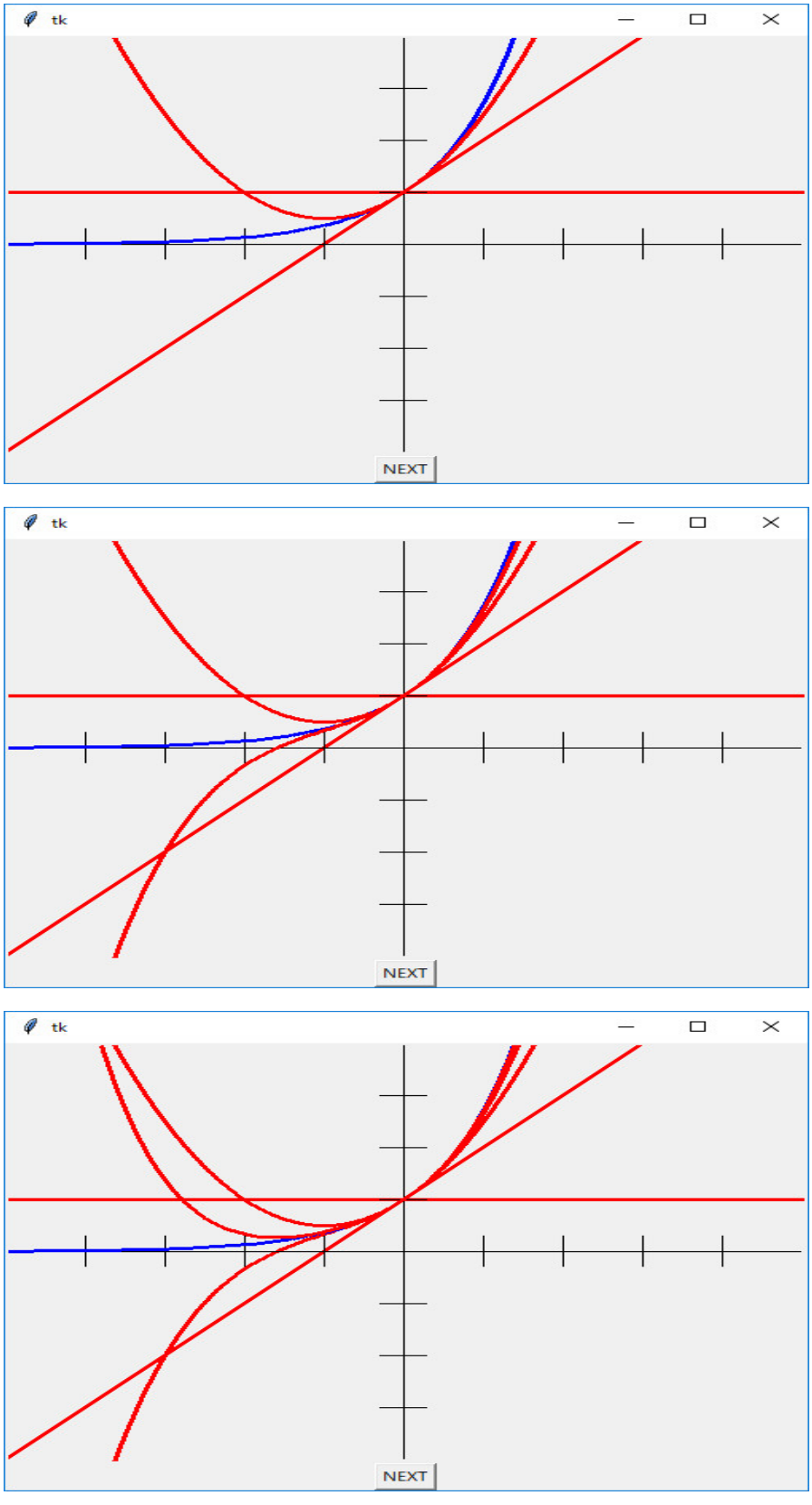
```
        r = r * i
    return r
def expN(n, x):
    s = 0.0
    for i in range(n+1):
        s = s + pow(x, i)/ fact(i)
    return s
def paint():
    global N
    n = N
    canvas.create_line(0, 200, 500, 200)
    canvas.create_line(250, 0, 250, 400)
    for x in range(0, 500, 50):
        canvas.create_line(x, 185, x, 215)
    for y in range(0, 400, 50):
        canvas.create_line(235, y, 265, y)
    x = -5
    dx = 0.1
    while x <= 5:
        canvas.create_line(250+50*x, 200-50*exp(x),
                           250+50*(x+dx), 200-50*exp(x+dx),
                           fill='blue', width=3.0)
        x = x + dx
    if n >= 0:
        x = -5
        dx = 0.1
        while x <= 5:
            canvas.create_line(250+50*x, 200-50*expN(n, x),
                               250+50*(x+dx), 200-50*expN(n, x+dx),
                               fill='red', width=3.0)
            x = x + dx
    N = N+1
root = Tk()
button = Button(root, text='NEXT', command=paint)
canvas = Canvas(root, width=500, height=400)
canvas.pack()
button.pack()
root.mainloop()
```

となります。実行し、NEXT ボタンをクリックすると順次、


tk
NEXT


tk
NEXT


tk
NEXT



と表示されます。y = expN(n; x) が y = exp(x) に近づいていく様子が見て取れます。paint の先

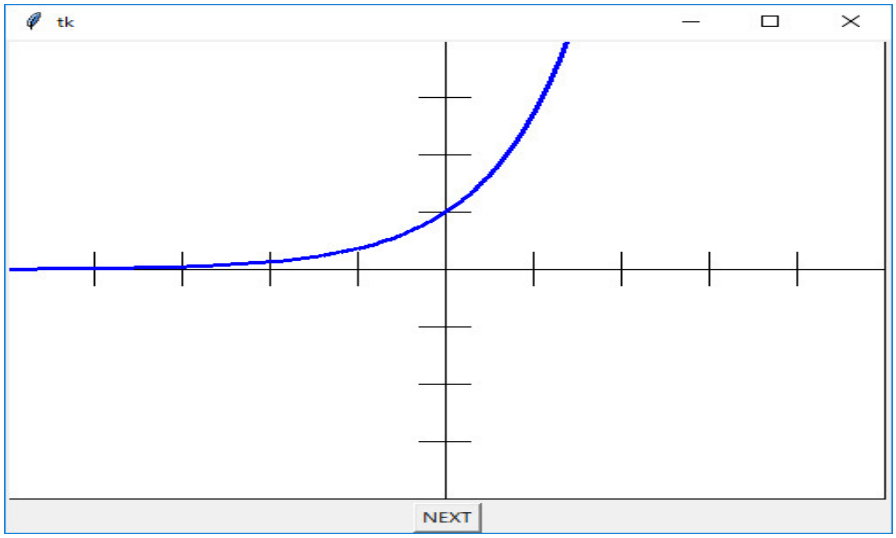
頭に

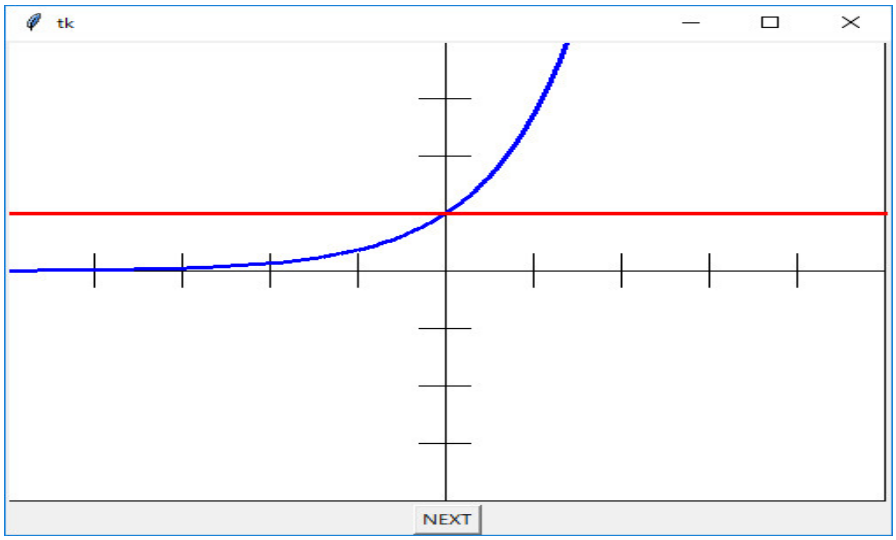
```
    canvas.create_rectangle(0, 0, 500, 400, fill='white')
```

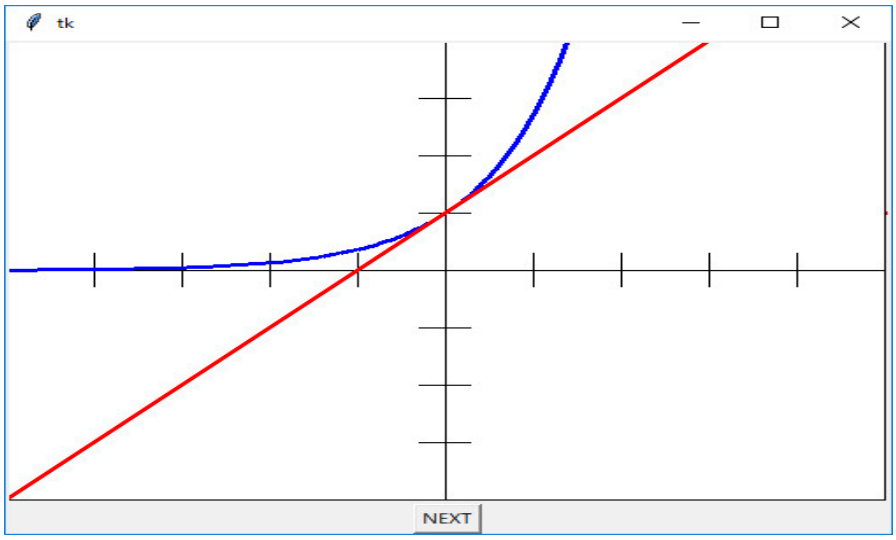
を追加して、paint を

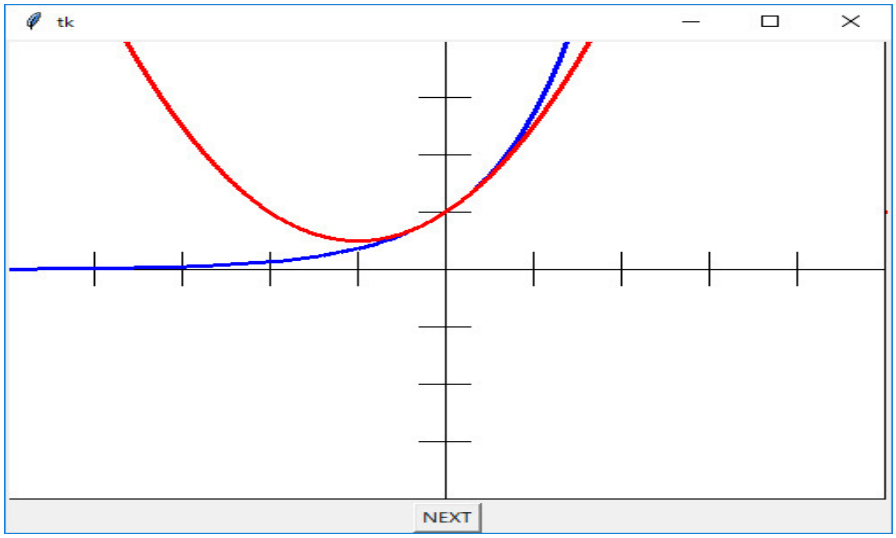
```
def paint():
    global N
    n = N
    canvas.create_rectangle(0, 0, 500, 400, fill='white')
    canvas.create_line(0, 200, 500, 200)
    canvas.create_line(250, 0, 250, 400)
    for x in range(0, 500, 50):
        canvas.create_line(x, 185, x, 215)
    for y in range(0, 400, 50):
        canvas.create_line(235, y, 265, y)
    x = -5
    dx = 0.1
    while x <= 5:
        canvas.create_line(250+50*x, 200-50*exp(x),
                           250+50*(x+dx), 200-50*exp(x+dx),
                           fill='blue', width=3.0)
        x = x + dx
    if n >= 0:
        x = -5
        dx = 0.1
        while x <= 5:
            canvas.create_line(250+50*x, 200-50*expN(n, x),
                               250+50*(x+dx), 200-50*expN(n, x+dx),
                               fill='red', width=3.0)
            x = x + dx


  N = N+1
```

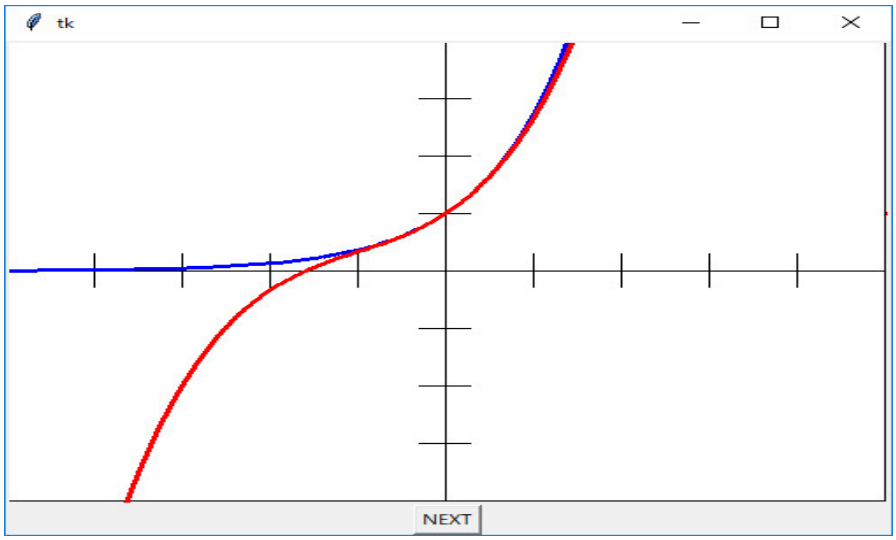
とすると

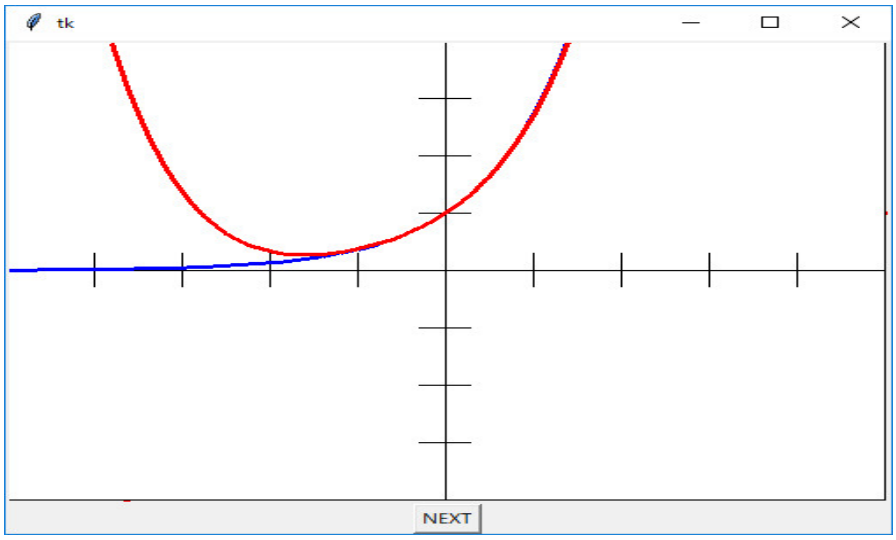

tk
NEXT

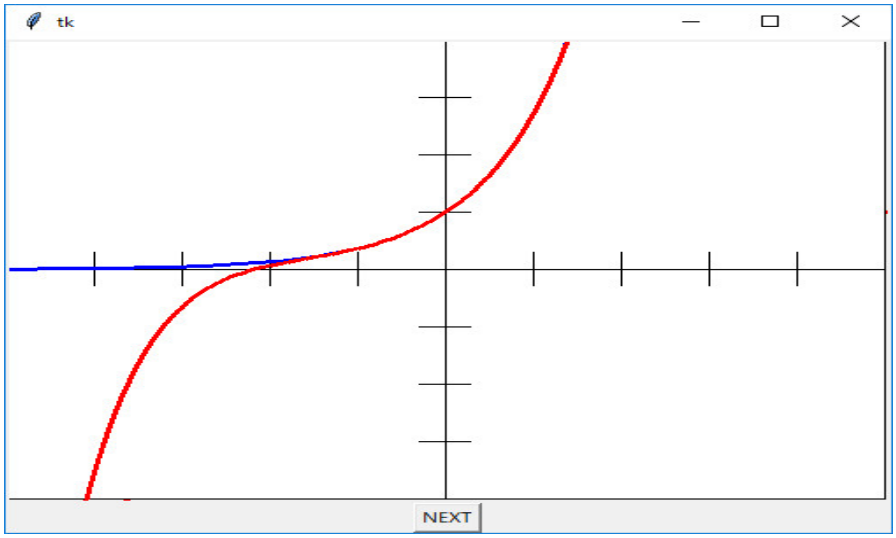

tk
NEXT


tk
NEXT


tk
NEXT


tk
NEXT


tk
NEXT



となる。

補足：階乗を計算する関数 $fact(n)$

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

は 4 つの部分から構成されています。関数は、

```
def    関数名 (引数の並び) :
    関数の本体
```

の構成になっています。関数名は fact です。引数の並びは今の場合、引数は一個で、引数 n があることを示しています。関数の本体は、字下げされた部分で、今の場合、

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

が関数の本体で、実数値を取る局所変数 r の宣言と初期設定 (r は関数の中で有効な局所変数であるという宣言とその値を 1.0 とするという初期設定が同時に行われています。値を 1 ではなく、1.0 とすることにより r が実数値を保持することが宣言されています) があって、次の for 文で、整数値の変数 i を 1 から n まで 1 つづつ増やしながら変化させ、(簡潔に言うと、整数値変数 i を 1 から n まで 1 つづつ増やしながら、r に順次掛けています。この結果、for 文が修了したとき、r は n の階乗の値になっています。最後に、r の値を関数の値として return 文で返しています。これで、階乗の手続き的定義

```
n! = 1 * 2 * 3 * ... * n
```

を Python で表現したことになります。これを Prolog や Scheme でやったように、再帰的に

```
def fact(n):
    if n == 0:
        return 1.0
    else:
        return n * fact(n-1)
```

と定義することも出来ます。勿論、

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

や

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

と整数値を返す関数として定義することも出来ます。if 文は、Scheme の場合と異なり、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 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。級数の最初の $n+1$ 項までの和を計算する関数 $expN(n,x)$

```
def expN(n, x):
    s = 0.0
    for i in range(n+1):
        s = s + pow(x, i) / fact(i)
    return s
```

も、関数名が expN で、引数の並びが n, x と 2 つの引数が宣言されています。今の場合、整数値を取る引数 n と実数値をとる引数 x が, で区切られて並んでいますが、定義上はその指定がありません。Python はインタプリッタなので、expN(n, x) が呼ばれた時点で、n には整数値が、x には実数値がセットされていることを Pyton は認識します。関数の本体は

```
    s = 0.0
    for i in range(n+1):
        s = s + pow(x, i) / fact(i)
    return s
```

で、実数値局所変数 s の宣言と初期設定（s は局所変数であるという宣言とその値を 0.0 とするという初期設定、これで s は実数値を保持するという宣言）がまずなされ、次に for 文で、整数値の変数 i を 0 から n まで、 1 つづつ増やしながら変化させ、s に x の i 乗を i の階乗で割った値を加えています。最後に、s の値を関数の値として return 文で返しています。これも Prolog や Scheme でやったように、再帰的に

```
def expN(n, x):
    if n == -1:
        return 0.0
```

```
    else:
        return pow(x, n) / fact(n) + expN(n-1, x)
```

と定義することも出来ます。さらに C++ のプログラムのように、関数 $pow(x, n)$ を使っていますが、Python ではこの関数は $x**n$ と表現することができます。しかも、整数乗だけでなく、$sqrt(x)$ を $x**0.5$ と表現することもできます。

ある程度以上のプログラムになると上のようにプログラミングがすらすら行くことはまずありません。例えば、range(n) の定義がうろ覚えであって、階乗を計算する関数 fact(n)

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

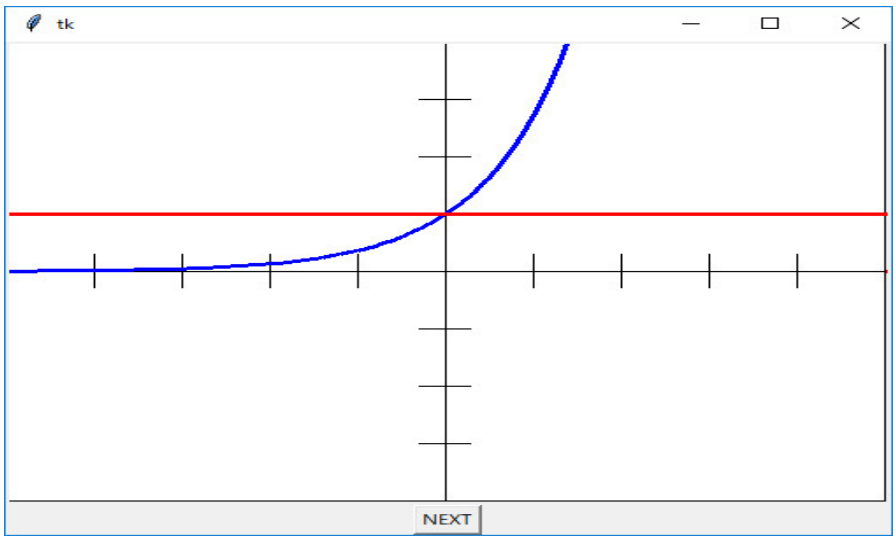
を間違って、

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

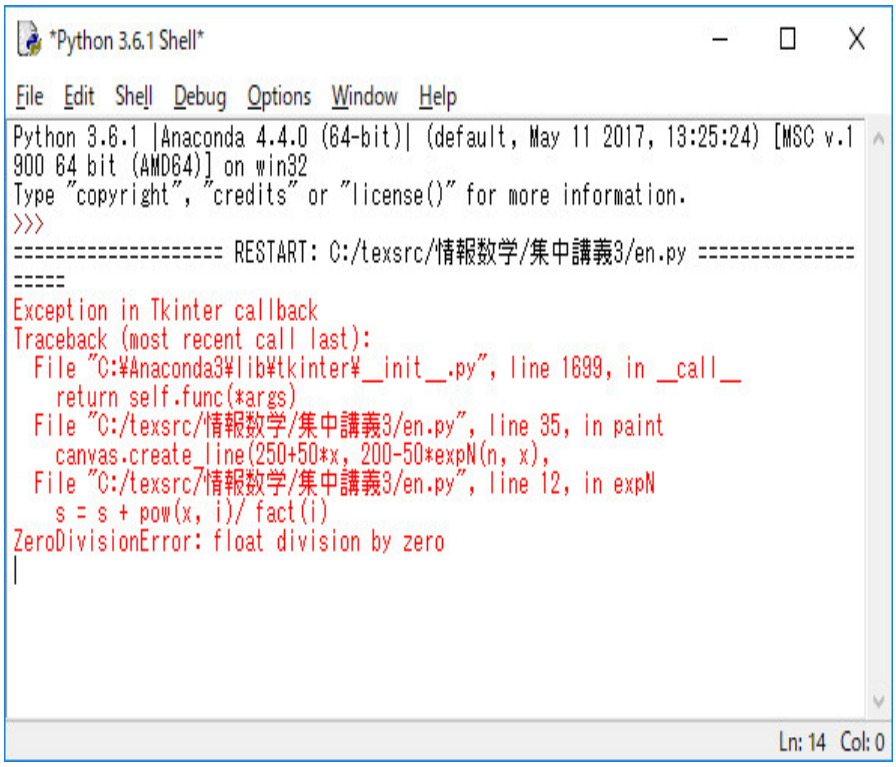
と定義し、expN(n, x) を

```
def expN(n, x):
    s = 0.0
    for i in range(n):
        s = s + pow(x, i) / fact(i)
    return s
```

たとします。NEXT ボタンをクリックしていくと



を表示した後、エラー表示が出ます。



最後に

```
File "D:\python\情報数学\exp.py", line 15, in expN
s = s + pow(x, i)/ fact(i)
ZeroDivisionError: float division by zero
```

と「ゼロで割った」というエラーが表示されています。fact(i) が 0 になったということです。fact(n) の定義を見直し、何故 0 になったか考えます。

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

と定義したのですが、range(n) は 0 から始まるリストなので、

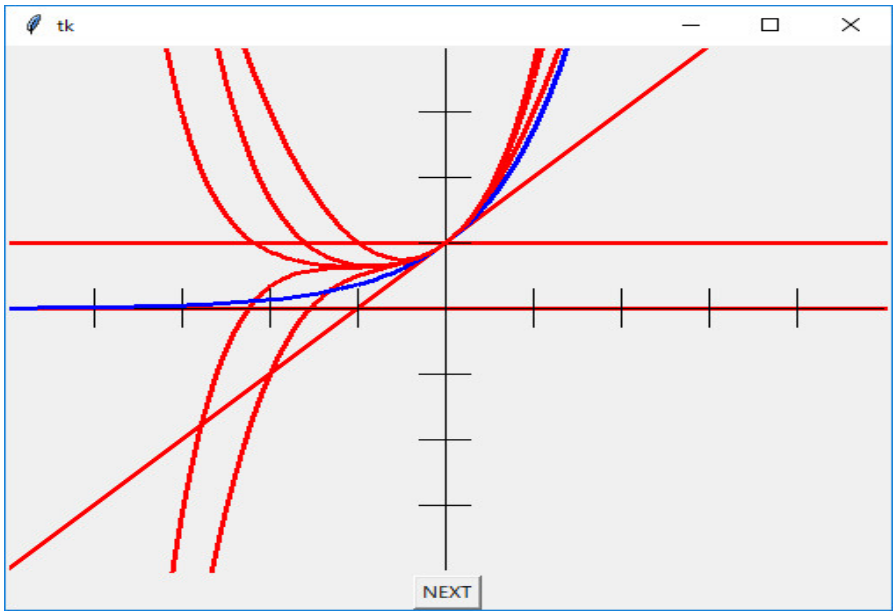
```
    for i in range(n):
        r = r * i
```

で、i は $0, 1, 2, \cdots$ と変化し、いつでもこの for 文を実行すると r = 0 となってしまいます。range(n) では、まず　i = 0 になることに、気が付いて、i=1 から始まるように、

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

```
    for i in range(1, n):
        r = r * i
    return r
```

と修正します



のように、$y = exe(x)$ に近づくはずの $y = expN(n; x)$ が $y = exe(x)$ に近づきません。まだ、間違っています。これは

```
    r = 1.0
    for i in range(1, n):
        r = r * i
```

では、$range(1, n) = [1, 2, 3, \cdots, n-1]$ で、$fact(n) = n!$ でなく、$fact(n) = (n-1)!$ になっています。この様な場合、思い込みが激しいのでプログラムリストを読み返すだけでは誤りを見つけられないことが多いです。さらに、実は $expN(n, x)$ も間違っています。

この様な場合には

```
    x = -1
    print( "fact(", n, ")=", fact(n))
    print( "expN(", n, ",", x, ")=", expN(n, x))
```

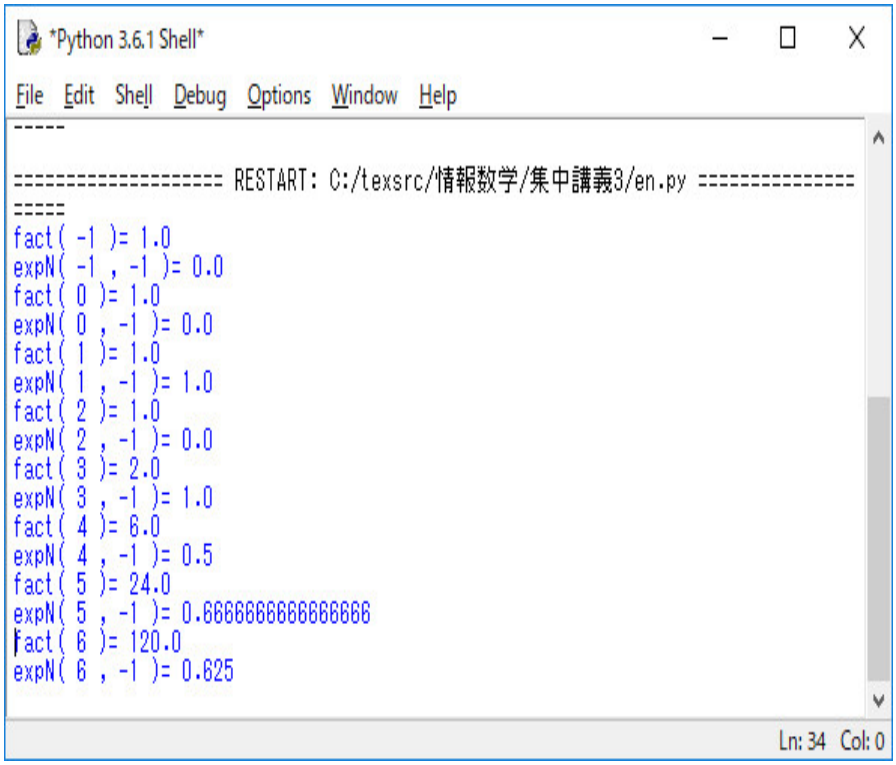
のような命令を追加して、

```
from tkinter import *
from math import *

N = -1

def fact(n):
```

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

def paint():
    global N
    n = N
    canvas.create_line(0, 200, 500, 200)
    canvas.create_line(250, 0, 250, 400)
    for x in range(0, 500, 50):
        canvas.create_line(x, 185, x, 215)
    for y in range(0, 400, 50):
        canvas.create_line(235, y, 265, y)
    x = -5
    dx = 0.1
    while x <= 5:
        canvas.create_line(250+50*x, 200-50*exp(x),
                           250+50*(x+dx), 200-50*exp(x+dx),
                           fill='blue', width=3.0)
        x = x + dx

    x = -1
    print( "fact(", n, ")=", fact(n))
    print( "expN(", n, ",", x, ")=", expN(n, x))

    if n >= 0:
        x = -5
        dx = 0.1
        while x <= 5:
            canvas.create_line(250+50*x, 200-50*expN(n, x),
                           250+50*(x+dx), 200-50*expN(n, x+dx),
                           fill='red', width=3.0)
            x = x + dx
    N = N+1

root = Tk()
button = Button(root, text='NEXT', command=paint)
```

```
canvas = Canvas(root, width=500, height=400)
canvas.pack()
button.pack()
root.mainloop()
```

というプログラムを実行してみます。NEXT ボタンを何回か押すと



のような表示が得られます。

```
>>> ================================ RESTART ================================
>>>
fact( -1 )= 1.0
expN( -1 , -1 )= 0.0
fact( 0 )= 1.0
expN( 0 , -1 )= 0.0
fact( 1 )= 1.0
expN( 1 , -1 )= 1.0
fact( 2 )= 1.0
expN( 2 , -1 )= 0.0
fact( 3 )= 2.0
expN( 3 , -1 )= 1.0
fact( 4 )= 6.0
expN( 4 , -1 )= 0.5
fact( 5 )= 24.0
expN( 5 , -1 )= 0.666666666667
```

```
fact( 6 )= 120.0
expN( 6 , -1 )= 0.625
>>>
```

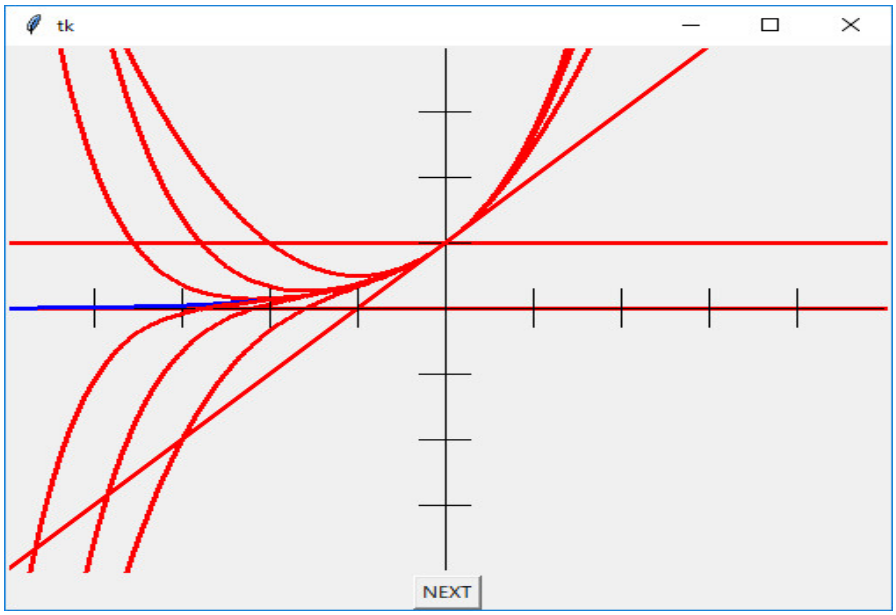
$fact(n)$ の値が一つずれていることに気が付きます。

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

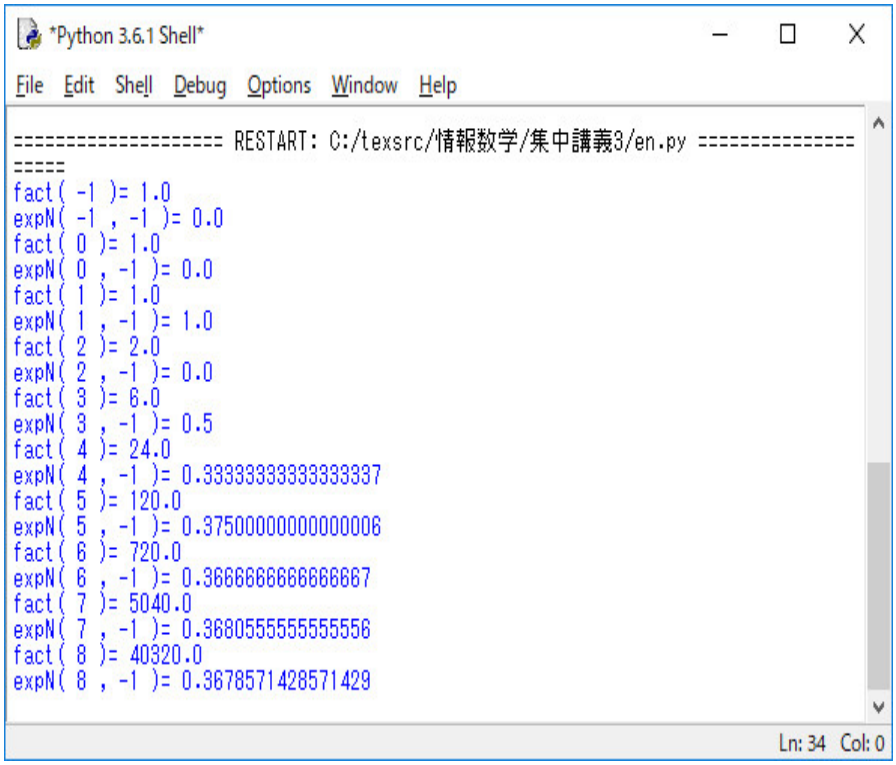
を見直し、range(n) を表示してみるなり、range(n) の定義をテキストやインターネットで調べて

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

と修正します。



のような表示が得られます。$y = exp(x)$ に近づくようになりましたが、最初に x 軸を塗り潰します。



のような表示です。

```
>>> ================================ RESTART ================================
>>>
fact( -1 )= 1.0
expN( -1 , -1 )= 0.0
fact( 0 )= 1.0
expN( 0 , -1 )= 0.0
fact( 1 )= 1.0
expN( 1 , -1 )= 1.0
fact( 2 )= 2.0
expN( 2 , -1 )= 0.0
fact( 3 )= 6.0
expN( 3 , -1 )= 0.5
fact( 4 )= 24.0
expN( 4 , -1 )= 0.333333333333
fact( 5 )= 120.0
expN( 5 , -1 )= 0.375
fact( 6 )= 720.0
expN( 6 , -1 )= 0.366666666667
fact( 7 )= 5040.0
expN( 7 , -1 )= 0.368055555556
fact( 8 )= 40320.0
expN( 8 , -1 )= 0.367857142857
```

```
>>>
```

$expN(n, x)$ も一つずれています。

```
def expN(n, x):
    s = 0.0
    for i in range(n):
        s = s + pow(x, i) / fact(i)
    return s
```

を

```
def expN(n, x):
    s = 0.0
    for i in range(n+1):
        s = s + pow(x, i) / fact(i)
    return s
```

と修正します。これで期待した図を描くようになりました。

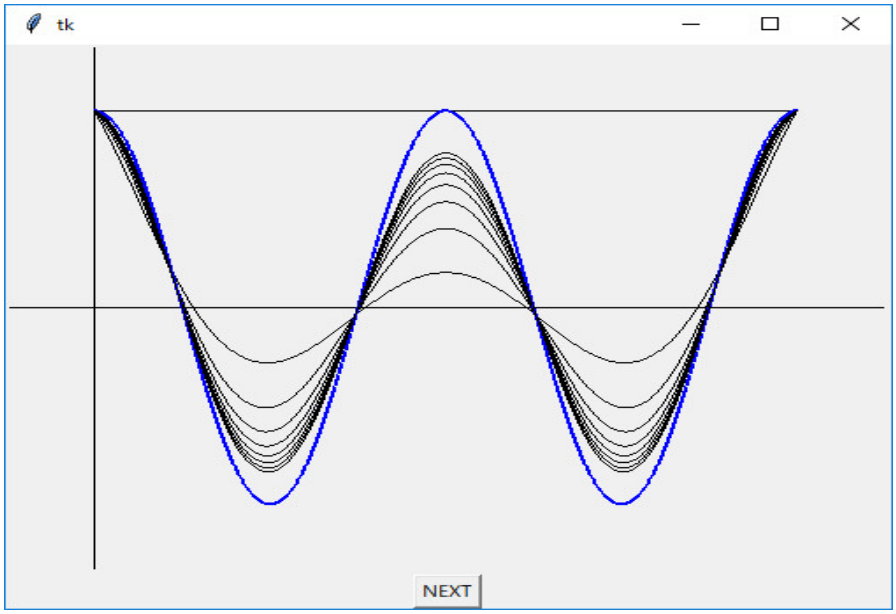
このようにプログラムの論理的間違いを見つけることを **debug** （**debugging**) と言います。この例のような場合は、用心深い人は関数を定義する度に、それを単独でチェックするプログラムを実行し、間違いがないか調べます。この様なことを習慣にしていれば上のような間違いは避けることが出来ますが、普通プログラムが上手く動かなければ、ここでやったように適当なところで、値を表示し、期待した通りにプログラムが実行されているか調べて見ます。何処で、どのような値を表示してみればよいかは勘と経験です。皆、言わないだけで、色々な失敗を繰り返し、試行錯誤しながら、**bug** を取り除き、少しづつプログラミング能力が上達します。間違いには実に色々な種類があって、間違えていそうなところを予想しながら、試行錯誤で途中の経過を観察し、次に何を調べるべきか考え続けなければいけないので、マンツーマンでも指導するのが難しいので、本には **debug** の方法がほとんど書いていませんが、この **debugging** の技術修得が本当は一番大切なことです。自分でプログラムの論理的間違いを見つけ修正する事が出来るようになれば、一人前です。多くの人がそれが出来ずに、プログラミングに挫折しますが、人は一番大切なことは普通教えてくれません。自分でその方法を見つけなければいけません。

同じ考えで出来る $cos4\pi x$ のベルンスタインの多項式による近似は

```
from tkinter import *
from math import *
def combi(n, k):
    r = 1.0
    for i in range(k):
        r = r * (n-i)/(i+1)
    return r
def g(n, k, x):
    r = combi(n,k) * x**k * (1-x)**(n-k) * cos(4*pi*float(k)/n)
    return r
def f (n, x):
    r = 0.0
```

```
    for k in range(n+1):
        r += g(n,k,x)
    return r
def paint():
    global N
    canvas.create_line(0, 200, 500, 200)
    canvas.create_line(50, 0, 50, 400)
    t = 0
    dt = 1.0/400
    while t < 1:
        canvas.create_line(50+400*t, 200-150*cos(4*pi*t),
            50+400*(t+dt), 200-150*cos(4*pi*(t+dt)), fill='blue', width=2.0)
        t = t + dt
    t = 0
    dt = 1.0/400
    while t < 1:
        canvas.create_line(50+400*t, 200-150*f(N, t),
            50+400*(t+dt), 200-150*f(N, t+dt))
        t = t + dt
    N = N+10
root = Tk()
N=2
button = Button(root, text='NEXT', command=paint)
canvas = Canvas(root, width=500, height=400)
canvas.pack()
button.pack()
root.mainloop()
```

でチェックすることが出来ます。実行すると



のような表示が得られます。関数 $f \in C[0,1]$ のベルンスタインの多項式とは

$$f_n(x) = \sum_{k=0}^{n} {}_nC_k x^k (1-x)^{n-k} f(\frac{k}{n})$$

のことです。この $f_n(x)$ は $f(x)$ に一様収束します。

次に**悪魔の階段**を描くプログラムを考察します。これは次のようにして定義される関数です。$x \in [0,1]$ を 3 進数に表現して、$x = 0.a_1a_2\cdots$ として、n を $a_n = 1$ となる初めての整数とします。このとき

$$F(x) = \sum_{1 \le m \le n, a_m \ge 1} \frac{1}{2^m}$$

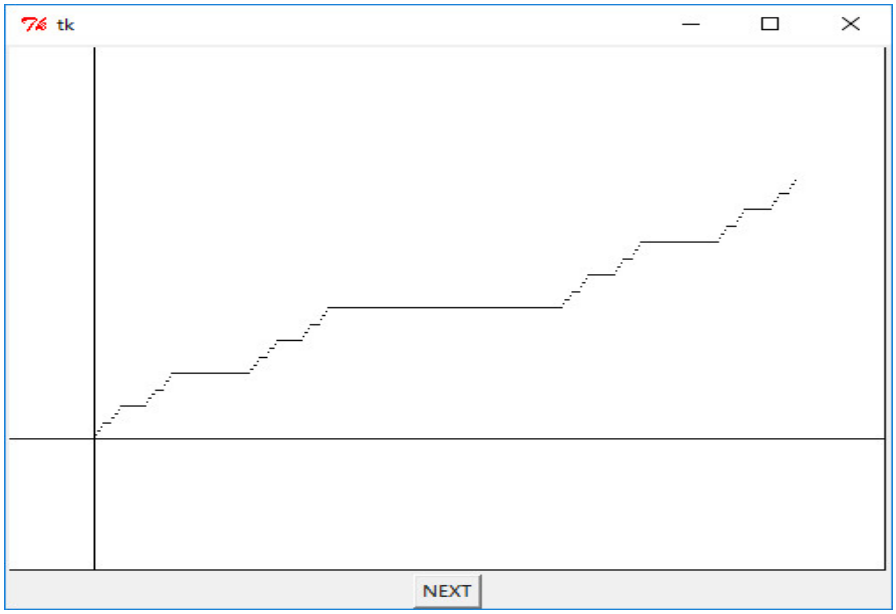
とおくと、この関数はどこかで 1 が現れる x について定義されますので、単調増加となります。この関数は $[0,1]$ の連続関数に拡張でき、カントール関数と呼びますが、ほとんどいたるところで微分が 0 なのに、$F(0) = 0, F(1) = 1$ と 0 から 1 まで単調に増加することから、悪魔の階段とも呼ばれます。

プログラムは

```
from tkinter import *
def tento3(i, N):
    r =[0]*N
    for k in range(N):
        r[k] = i % 3
        i = i / 3
    return r
def f(i, N):
    series = tento3(i, N)
    t = 0.0
    for k in range(N):
```

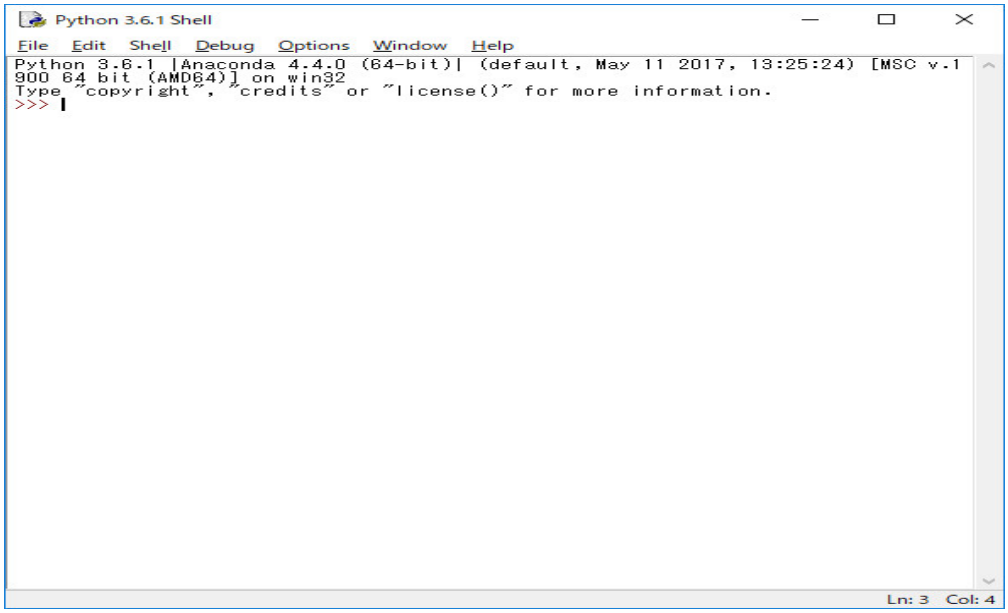
```
        if series[N-k-1] == 1:
            t += (1.0/2)**(k+1)
            break
        elif series[N-k-1] == 2:
            t += (1.0/2)**(k+1)
    return t
def paint():
    global N
    canvas.create_rectangle(0, 0, 500, 400, fill='white')
    canvas.create_line(0, 300, 500, 300)
    canvas.create_line(50, 0, 50, 400)
    t = 0
    dt = 1.0 / (3**N)
    for i in range(3**N):
        canvas.create_line(50+400*t, 300-200*f(i, N),
                           50+400*(t+dt), 300-200*f(i, N))
        t = t + dt
    N = N+1
root = Tk()
N=1
button = Button(root, text='NEXT', command=paint)
canvas = Canvas(root, width=500, height=400)
canvas.pack()
button.pack()
root.mainloop()
```

です。実行すると

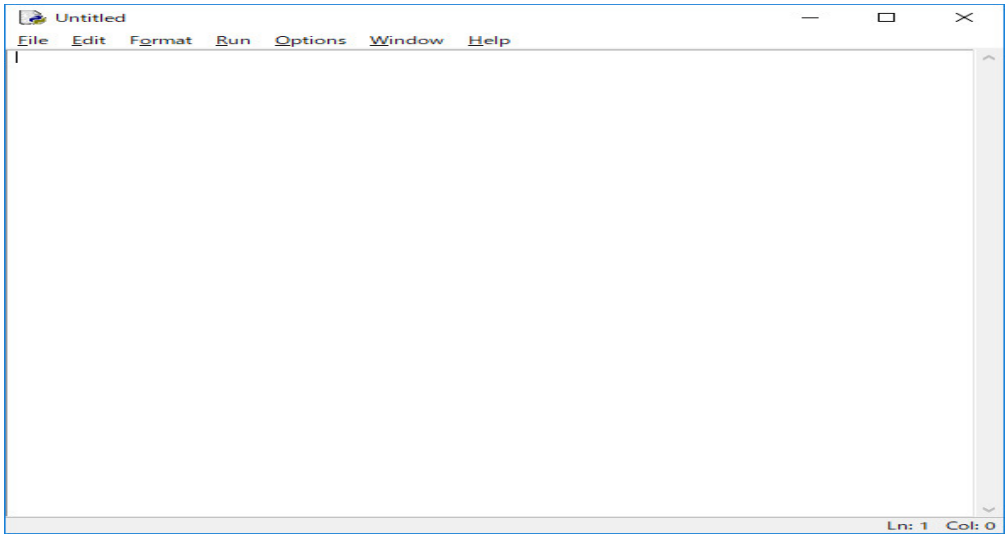


のような表示が得られます。

演習問題：　関数 $y = \sin(x) + \sin(2x)/2 + \sin(3x)/3$ を描くプログラムを作成しなさい。
演習問題：　$\sin(x)$ や $\cos(x)$ で同じ事をするプログラムを作成しなさい。
極方程式 r = 3sin(3 θ) のグラフを描いてみよう。
Python Shell を立ち上げ、



File メニューの　New File を選択し、Editor を開きます。



ここに、次のように打ち込む。

```
from tkinter import *
root = Tk()
canvas = Canvas(root, width = 350, height=350)
canvas.pack()
root.mainloop()
```

極方程式 $r = 3sin(3\ \theta\ )$ は θ を媒介変数として、$x(\ \theta\ ) = 3sin(3\ \theta\ )cos(\ \theta\ )$, $y(\ \theta\ ) = 3sin(3\ \theta\ )sin(\ \theta\ )$ と表されるから、

```
from math import *
def X(t) :
    return 3*sin(3*t)*cos(t)
def Y(t) :
    return 3*sin(3*t)*sin(t)
```

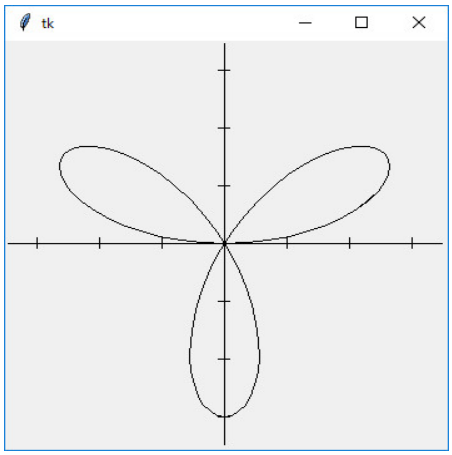
という関数の定義をプログラムの先頭に置き、canvas の定義の後に

```
canvas.create_line(0, 175, 350, 175)
canvas.create_line(175, 0, 175, 350)
for x in range(25, 350, 50):
    canvas.create_line(x, 170, x, 180)
for y in range(25, 350, 50):
    canvas.create_line(170, y, 180, y)
t = 0.0
dt = pi/80
while t < 2*pi:
    canvas.create_line(175+50*X(t), 175-50*Y(t), 175+50*X(t+dt), 175-50*Y(t+dt))
    t = t + dt
```

, と打ち込みます。プログラム全体は

```
from tkinter import *
from math import *
def X(t) :
    return 3*sin(3*t)*cos(t)
def Y(t) :
    return 3*sin(3*t)*sin(t)
root = Tk()
canvas = Canvas(root, width = 350, height=350)
canvas.create_line(0, 175, 350, 175)
canvas.create_line(175, 0, 175, 350)
for x in range(25, 350, 50):
    canvas.create_line(x, 170, x, 180)
for y in range(25, 350, 50):
    canvas.create_line(170, y, 180, y)
t = 0.0
dt = pi/80
while t < 2*pi:
    canvas.create_line(175+50*X(t), 175-50*Y(t), 175+50*X(t+dt), 175-50*Y(t+dt))
    t = t + dt
canvas.pack()
root.mainloop()
```

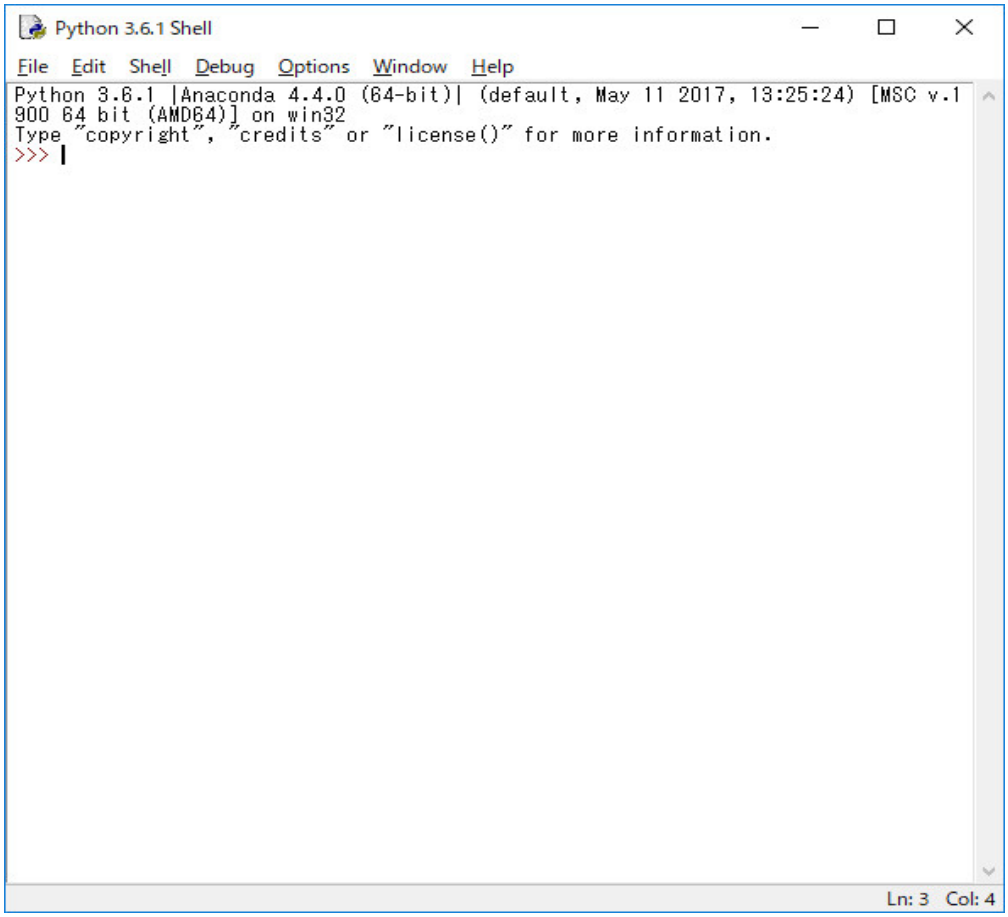
となります。プログラムに名前を付け、実行すると



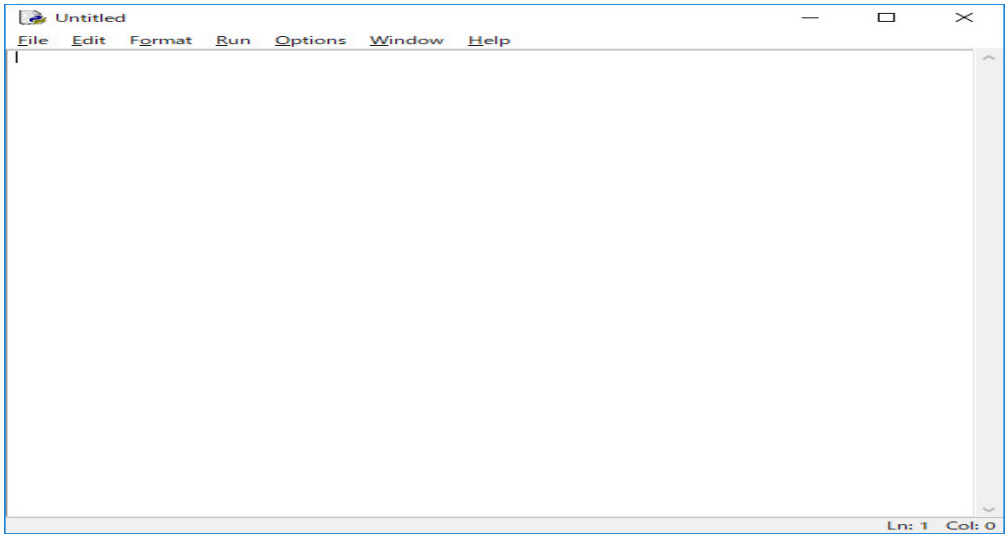
を描きます。

例題：$(1/2, 0)$ を中心とする直径 1 の円上の点を動点として、直径 OP の円群を描け。輪郭線にカージオイドが浮かび上がる。

Python Shell を立ち上げ、



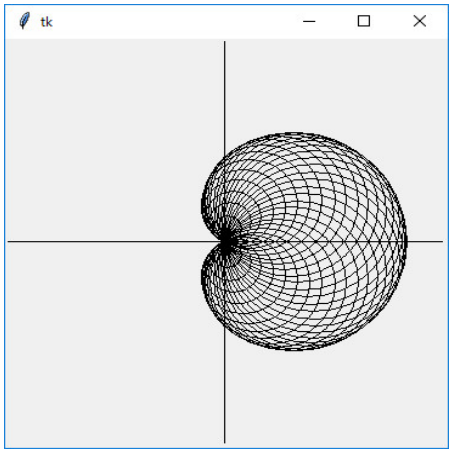
File メニューの　New File を選択し、Editor を開きます。



ここに、次のように打ち込む。

```
from tkinter import *
from math import *
root = Tk()
canvas = Canvas(root, width = 360, height=360)
canvas.create_line(0, 180, 360, 180)
canvas.create_line(180, 0, 180, 360)
K = 150
t = 0.0
while t <= 2*pi :
    x = 0.25+cos(t)/4
    y = sin(t)/4
    x1 = K*(x-sqrt(0.5+cos(t)/2)/2)+180
    y1 = K*(-y-sqrt(0.5+cos(t)/2)/2)+180
    x2 = x1 + K*sqrt(0.5+cos(t)/2)
    y2 = y1 + K*sqrt(0.5+cos(t)/2)
    canvas.create_oval(x1, y1, x2, y2)
    t = t + pi/20
canvas.pack()
root.mainloop()
```

プログラムに名前を付け、実行すると

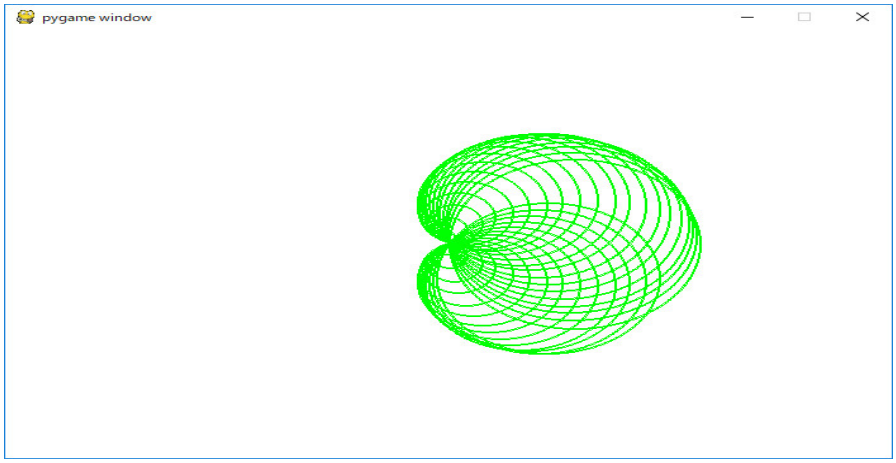


を描きます。

この図を描くアニメーションの Python and Pygame のプログラムは次のようになります。

```
import pygame
from math import *

pygame.init()

black = (0,0,0)
white = (255, 255, 255)
green = (0, 255, 0)
red= (255, 0, 0)

pi = 3.141592653
size = (700, 500)

screen = pygame.display.set_mode(size)

pi = atan(1.0)*4
done = False
clock = pygame.time.Clock()
theta = pi
K = 200

while done == False:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True

    screen.fill(white)
    t = 0
```

```
    while t <= 2*pi - 2*theta:
        x = 0.25 + cos(t)/4.0
        y = sin(t)/4.0
        x1 = (int)(K * ( x - sqrt(0.5+cos(t)/2.0)/2.0) + 350)
        y1 = (int)(K * (-y - sqrt(0.5+cos(t)/2.0)/2.0) + 250)
        x2 = (int)(K * sqrt(0.5+cos(t)/2.0))
        y2 = (int)(K * sqrt(0.5+cos(t)/2.0))
        if x2 == 0 or y2 == 0:
            t += pi/20
            continue
        pygame.draw.ellipse(screen, green, [x1, y1, x2, y2],2)
        t += pi/20
    theta -= pi/20

    pygame.display.flip()

    if theta < 0:
        theta = pi
    clock.tick(5)

pygame.quit()
```



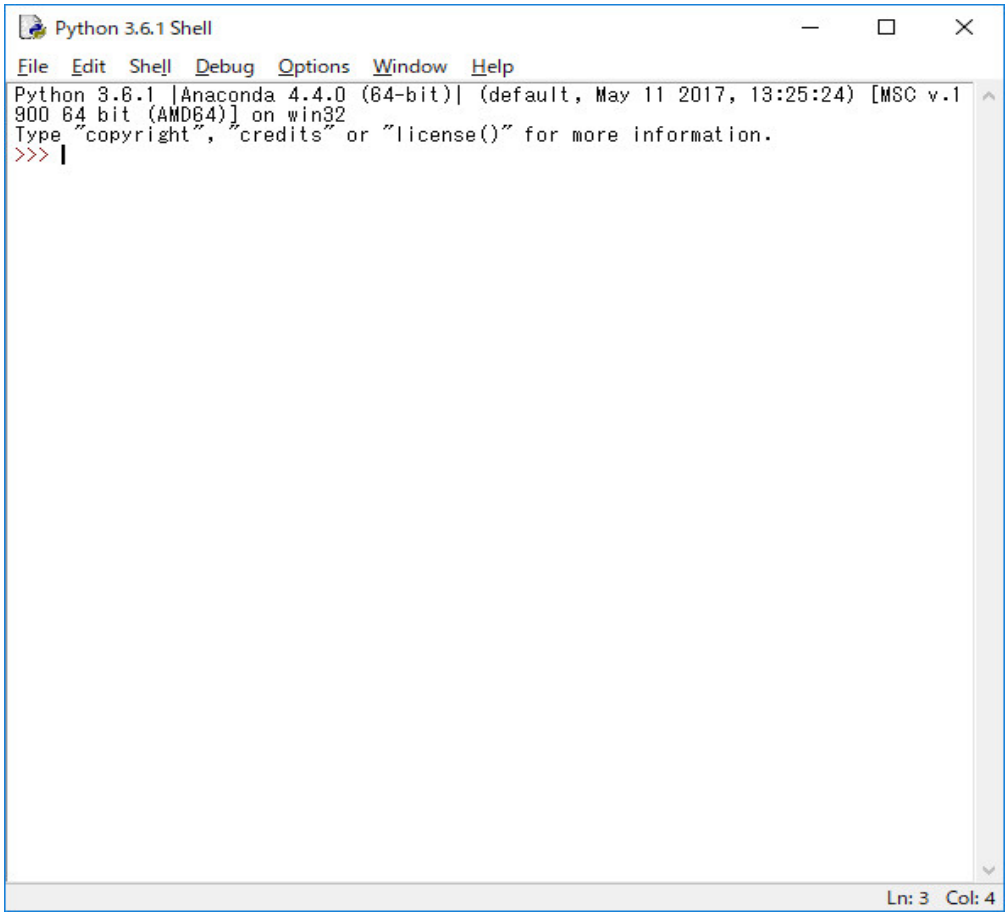
例題：原点を中心とし半径が a の円 O の外側を半径が b の円 C が円 O に外接しながら滑ることなく転がるとき、円 C 上の点 P の軌跡を考える。ただし、点 P のはじめの位置は、円 O と x 軸の正の部分との交点 A とする。

円 C が転がるとき、動径 OC が表す角を θ とし、そのときの外接する点を Q、点 P の座標を (x,y) とすると、P の軌跡は θ を媒介変数として

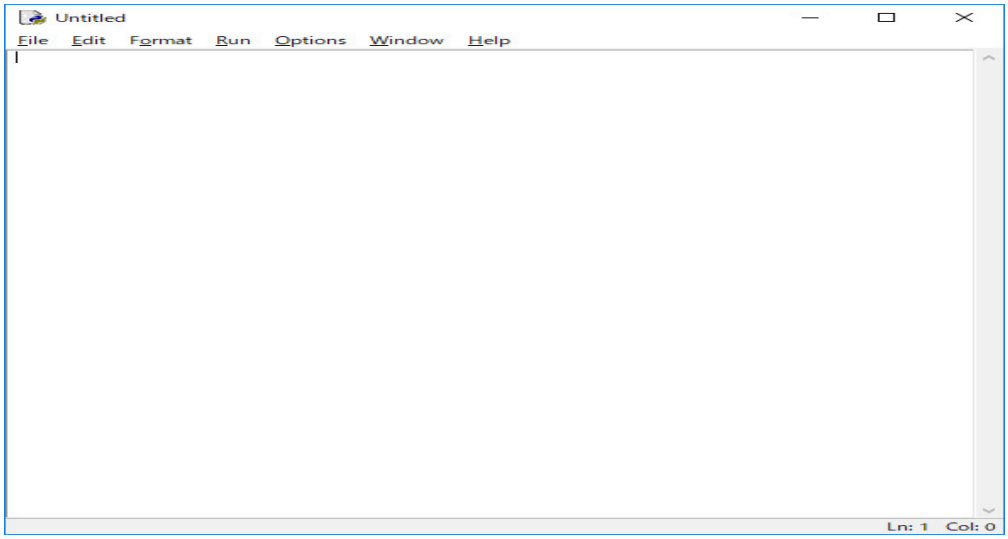
$$x = (a+b)\cos\theta - b\cos\frac{a+b}{b}\theta, \quad y = (a+b)\sin\theta - b\sin\frac{a+b}{b}\theta$$

で表される。この曲線を **エピサイクロイド** という。エピサイクロイドを描くプログラムを作れ。

Python Shell を立ち上げ、



File メニューの　New File を選択し、Editor を開きます。

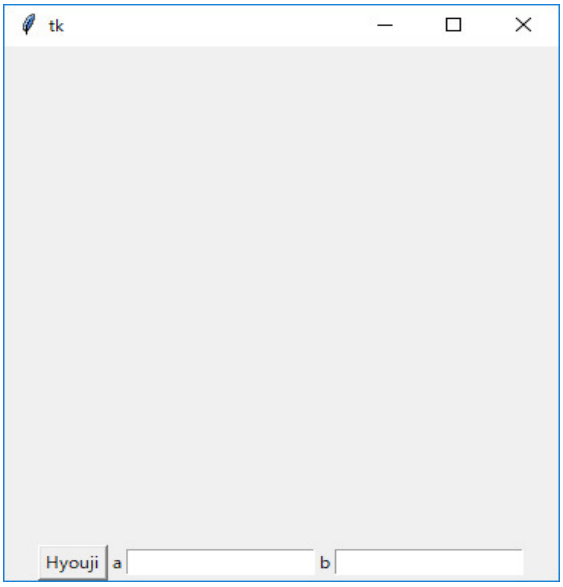


ここに、次のように打ち込む。

```
from tkinter import *
root = Tk()
```

```
f0 = Frame(root)
f1 = Frame(root)
canvas = Canvas(f0, width = 360, height=360)
canvas.pack()
button = Button(f1, text='Hyouji').pack(side = LEFT)
l1 = Label(f1, text='a').pack(side = LEFT)
t1 = Entry(f1).pack(side = LEFT)
l2 = Label(f1, text='b').pack(side = LEFT)
t2 = Entry(f1).pack(side = LEFT)
f0.pack()
f1.pack()
root.mainloop()
```

実行してみる。



中央に canvas、下辺にボタンと a と b と書かれたラベル 2 個と数値を入力するための Entry 2 個が配置されています。

ボタンを押せばグラフを描くようにします。Button の宣言で command オプションを追加します。つまり

```
button = Button(f1, text='Hyouji').pack(side = LEFT)
```

を

```
button = Button(f1, text='Hyouji', command=paint).pack(side = LEFT)
```

に変えます。そして、Entry に入力した数値を取り出せるように Entry の宣言を修正します。

```
from tkinter import *
root = Tk()
f0 = Frame(root)
```

```
f1 = Frame(root)
canvas = Canvas(f0, width = 360, height=360)
canvas.pack()
button = Button(f1, text='Hyouji', command=paint).pack(side = LEFT)
l1 = Label(f1, text='a').pack(side = LEFT)
content1 = StringVar()
t1 = Entry(f1, textvariable=content1).pack(side = LEFT)
l2 = Label(f1, text='b').pack(side = LEFT)
content2 = StringVar()
t2 = Entry(f1, textvariable=content2).pack(side = LEFT)
f0.pack()
f1.pack()
root.mainloop()
```

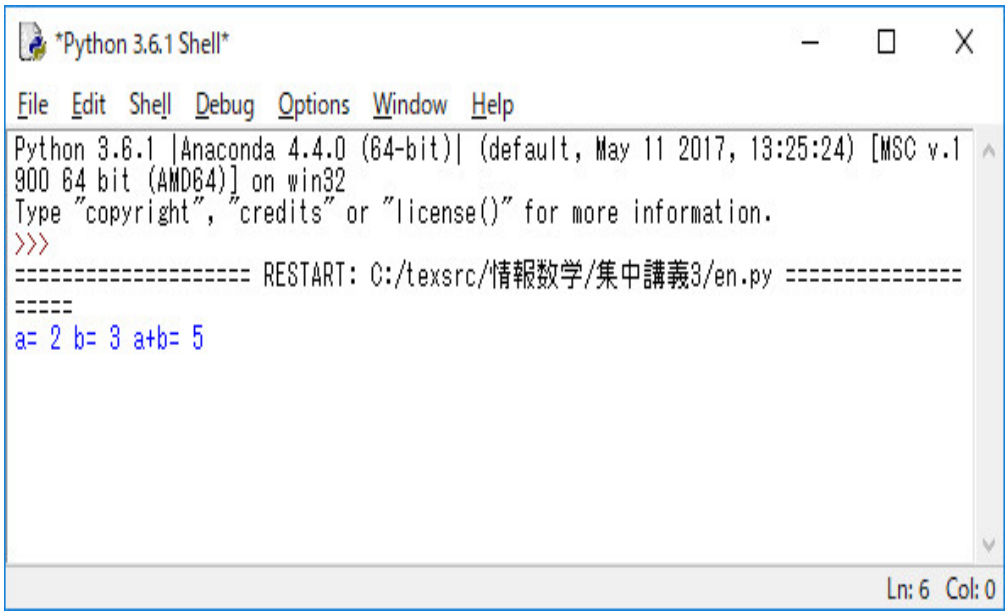
そして、まづは次のような関数 paint を定義します。

```
def paint():
    a = int(content1.get())
    b = int(content2.get())
    print( "a=", a, "b=", b, "a+b=", a+b )
```

結局、全体のプログラムは

```
from tkinter import *

def paint():
    a = int(content1.get())
    b = int(content2.get())
    print( "a=", a, "b=", b, "a+b=", a+b )

root = Tk()
f0 = Frame(root)
f1 = Frame(root)
canvas = Canvas(f0, width = 360, height=360)
canvas.pack()
button = Button(f1, text='Hyouji', command=paint).pack(side = LEFT)
l1 = Label(f1, text='a').pack(side = LEFT)
content1 = StringVar()
t1 = Entry(f1, textvariable=content1).pack(side = LEFT)
l2 = Label(f1, text='b').pack(side = LEFT)
content2 = StringVar()
t2 = Entry(f1, textvariable=content2).pack(side = LEFT)
f0.pack()
f1.pack()
root.mainloop()
```

となりました。実行して、Entry1 に 2 を Entry2 に 3 を入力して Hyouji のボタンをクリックします。

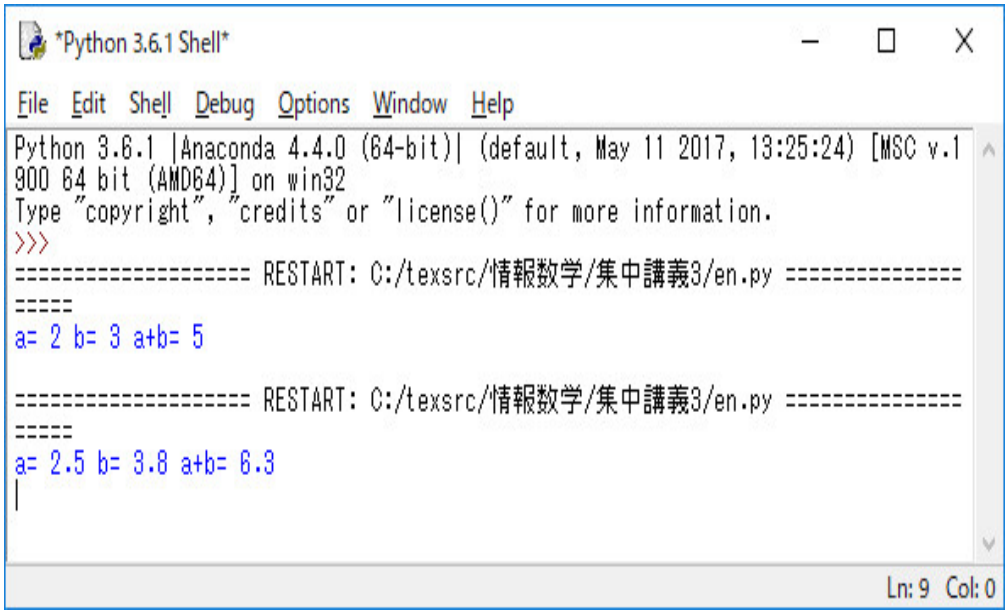



```
>>> ================================ RESTART ================================
>>>
a= 2 b= 3 a+b= 5
>>>
```

と整数値として取り出すことが出来ました。Entry1 に 2.5 を Entry2 に 3.8 を入力して数値として取り出すためには paint を次のように修正します。

```
def paint():
    a = float(content1.get())
    b = float(content2.get())
    print( "a=", a, "b=", b, "a+b=", a+b )
```

実行して、Entry1 に 2.5 を Entry2 に 3.8 を入力して Hyouji のボタンをクリックします。




```
>>> ================================ RESTART ================================
>>>
a= 2.5 b= 3.8 a+b= 6.3
>>>
```

と実数値として取り出すことが出来ました。

三角関数を使うので

```
from math import *
```

を先頭に追加し

```
from tkinter import *
from math import *

def paint():
    a = float(content1.get())
    b = float(content2.get())
    print "a=", a, "b=", b, "a+b=", a+b

root = Tk()
f0 = Frame(root)
f1 = Frame(root)
canvas = Canvas(f0, width = 360, height=360)
canvas.pack()
button = Button(f1, text='Hyouji', command=paint).pack(side = LEFT)
l1 = Label(f1, text='a').pack(side = LEFT)
content1 = StringVar()
t1 = Entry(f1, textvariable=content1).pack(side = LEFT)
```

```
l2 = Label(f1, text='b').pack(side = LEFT)
content2 = StringVar()
t2 = Entry(f1, textvariable=content2).pack(side = LEFT)
f0.pack()
f1.pack()
root.mainloop()
```

以下これを雛形としてプログラミングすれば良いです。

入力された a と b の値をもとにエピサイクロイドを描くように paint を定義します。

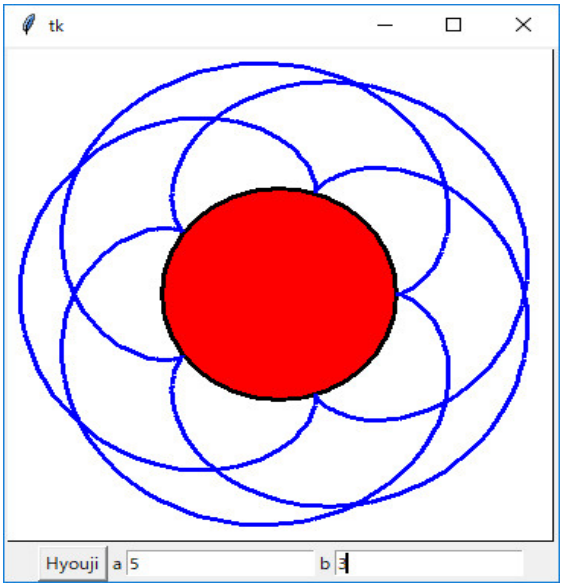
```
def paint():
    a = float(content1.get())
    b = float(content2.get())
    K = 170/(a+2*b)
    canvas.create_rectangle(0, 0, 360, 360, fill='white')
    canvas.create_oval(180-K*a, 180-K*a, 180+K*a, 180+K*a, fill='red', width=3.0)
    t = 0.0
    dt = pi/80
    while t <= 2*pi*b :
        x1 = 180+K*((a+b)*cos(t)-b*cos((a+b)*t/b))
        y1 = 180-K*((a+b)*sin(t)-b*sin((a+b)*t/b))
        x2 = 180+K*((a+b)*cos(t+dt)-b*cos((a+b)*(t+dt)/b))
        y2 = 180-K*((a+b)*sin(t+dt)-b*sin((a+b)*(t+dt)/b))
        canvas.create_line(x1, y1, x2, y2, fill='blue', width=3.0)
        t = t + dt
```

全体のプログラムは

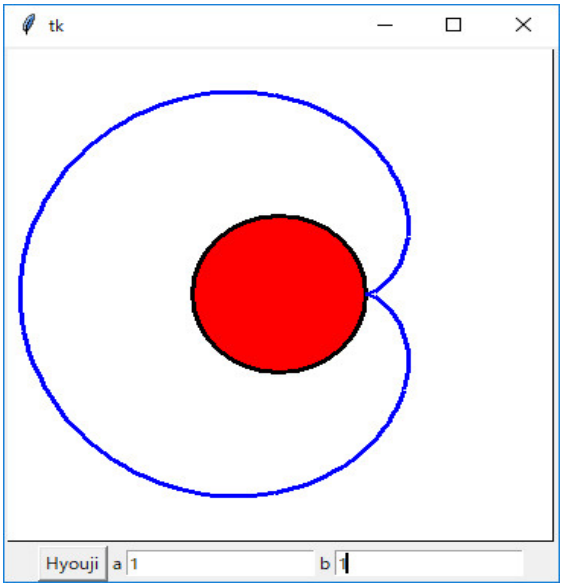
```
from tkinter import *
from math import *

def paint():
    a = float(content1.get())
    b = float(content2.get())
    K = 170/(a+2*b)
    canvas.create_rectangle(0, 0, 360, 360, fill='white')
    canvas.create_oval(180-K*a, 180-K*a, 180+K*a, 180+K*a, fill='red', width=3.0)
    t = 0.0
    dt = pi/80
    while t <= 2*pi*b :
        x1 = 180+K*((a+b)*cos(t)-b*cos((a+b)*t/b))
        y1 = 180-K*((a+b)*sin(t)-b*sin((a+b)*t/b))
        x2 = 180+K*((a+b)*cos(t+dt)-b*cos((a+b)*(t+dt)/b))
        y2 = 180-K*((a+b)*sin(t+dt)-b*sin((a+b)*(t+dt)/b))
        canvas.create_line(x1, y1, x2, y2, fill='blue', width=3.0)
```

```
        t = t + dt

root = Tk()
f0 = Frame(root)
f1 = Frame(root)
canvas = Canvas(f0, width = 360, height=360)
canvas.pack()
button = Button(f1, text='Hyouji', command=paint).pack(side = LEFT)
l1 = Label(f1, text='a').pack(side = LEFT)
content1 = StringVar()
t1 = Entry(f1, textvariable=content1).pack(side = LEFT)
l2 = Label(f1, text='b').pack(side = LEFT)
content2 = StringVar()
t2 = Entry(f1, textvariable=content2).pack(side = LEFT)
f0.pack()
f1.pack()
root.mainloop()
```

となります。実行して $a = 5, b = 3$ とすると

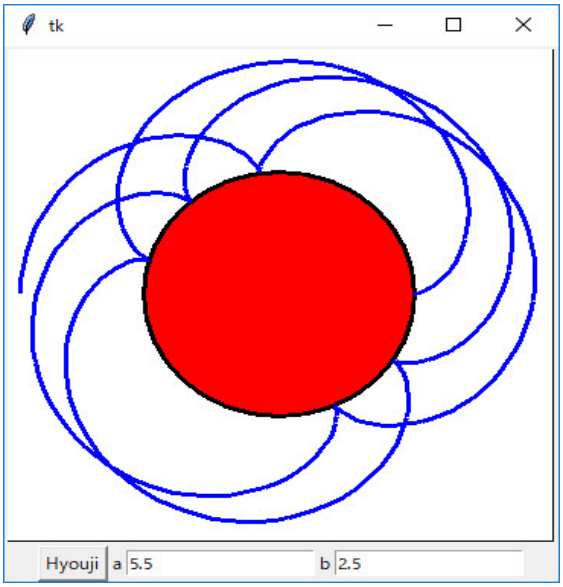


となります。実行して $a = 1, b = 1$ とすると

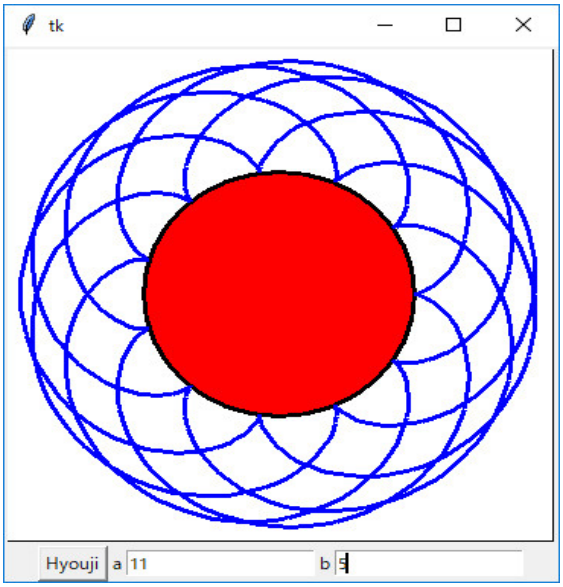


となります。

実行して a = 5.5, b = 2.5 とすると



となります。二倍して a = 11. b = 5 とすると

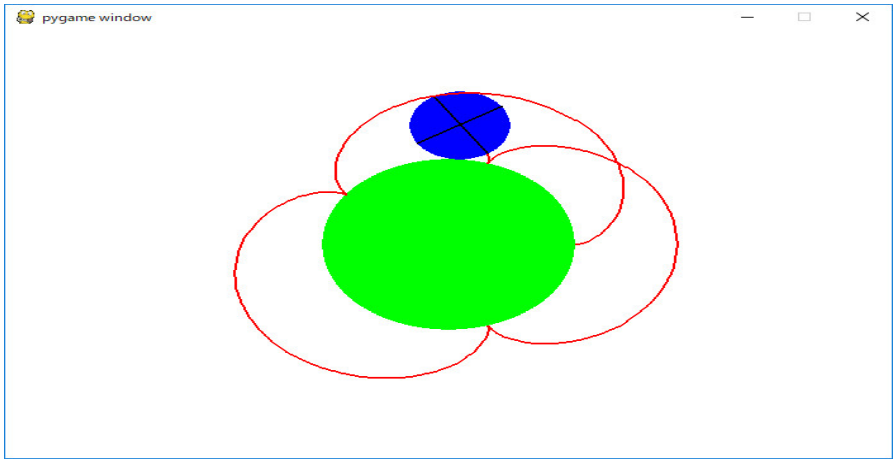


となります。折角、実数値を入力できるようにしましたが整数値でないとうまく動きません。
　この図を描く様子をアニメーションで見るには

```
import pygame
from math import *

pygame.init()

black = (0,0,0)
white = (255, 255, 255)
green = (0, 255, 0)
red= (255, 0, 0)
blue = (0, 0, 255)

pi = 3.141592653
size = (700, 500)

screen = pygame.display.set_mode(size)

pi = atan(1.0)*4
done = False
clock = pygame.time.Clock()
theta = 0
K = 20
a = 5
b = 2

while done == False:
```

```
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True

    screen.fill(white)
    pygame.draw.ellipse(screen, green, [350-K*a, 250-K*a, 2*K*a, 2*K*a])
    x = (int)(K*((a+b)*cos(theta)-b))
    y = (int)(K*((a+b)*sin(theta)+b))
    pygame.draw.ellipse(screen, blue,[350+x,250-y, 2*K*b, 2*K*b])
    x1 = (int)(350+K*((a+b)*cos(theta)-b*cos((a+b)*theta/b)))
    y1 = (int)(250-K*((a+b)*sin(theta)-b*sin((a+b)*theta/b)))
    x2 = (int)(350+K*((a+b)*cos(theta)+b*cos((a+b)*theta/b)))
    y2 = (int)(250-K*((a+b)*sin(theta)+b*sin((a+b)*theta/b)))
    pygame.draw.line(screen, black, [x1, y1], [x2, y2], 2)
    x1 = (int)(350+K*((a+b)*cos(theta)-b*cos((a+b)*theta/b+pi/2)))
    y1 = (int)(250-K*((a+b)*sin(theta)-b*sin((a+b)*theta/b+pi/2)))
    x2 = (int)(350+K*((a+b)*cos(theta)+b*cos((a+b)*theta/b+pi/2)))
    y2 = (int)(250-K*((a+b)*sin(theta)+b*sin((a+b)*theta/b+pi/2)))
    pygame.draw.line(screen, black, [x1, y1], [x2, y2], 2)

    t = 0
    ox = (int)(350+K*((a+b)*cos(t)-b*cos((a+b)*t/b)))
    oy = (int)(250-K*((a+b)*sin(t)-b*sin((a+b)*t/b)))
    while t <= theta:
        nx = (int)(350+K*((a+b)*cos(t)-b*cos((a+b)*t/b)))
        ny = (int)(250-K*((a+b)*sin(t)-b*sin((a+b)*t/b)))
        pygame.draw.line(screen, red, [ox, oy], [nx, ny], 2)
        ox = nx
        oy = ny
        t += pi/100

    theta += pi/25
    if theta >= 2*pi*b:
        theta = 0
    pygame.display.flip()

    clock.tick(5)

pygame.quit()
```

とすればよい。

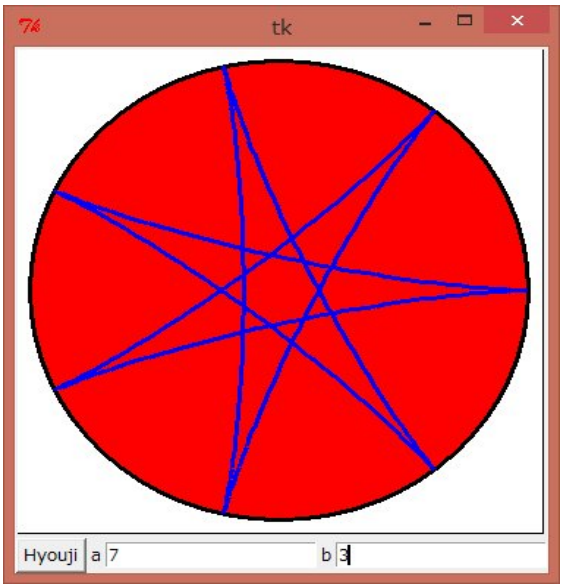


レポート問題：原点を中心とし半径が **a** の円 **O** の内側を半径が **b** $a > b > 0)$ の円 **C** が円 **O** に内接しながら滑ることなく転がるとき、円 **C** 上の点 **P** の軌跡を考える。ただし、点 **P** のはじめの位置は、円 **O** と **x** 軸の正の部分との交点 **A** とする。

円 **C** が転がるとき、動径 **OC** が表す角を θ とし、そのときの外接する点を **Q**、点 **P** の座標を **(x,y)** とすると、**P** の軌跡は θ を媒介変数として

$$x = (a-b)\cos\theta + b\cos\frac{a-b}{b}\theta, \quad y = (a-b)\sin\theta - b\sin\frac{a-b}{b}\theta$$

で表される。この曲線を ハイポサイクロイド という。ハイポサイクロイドを描くプログラムを作れ。



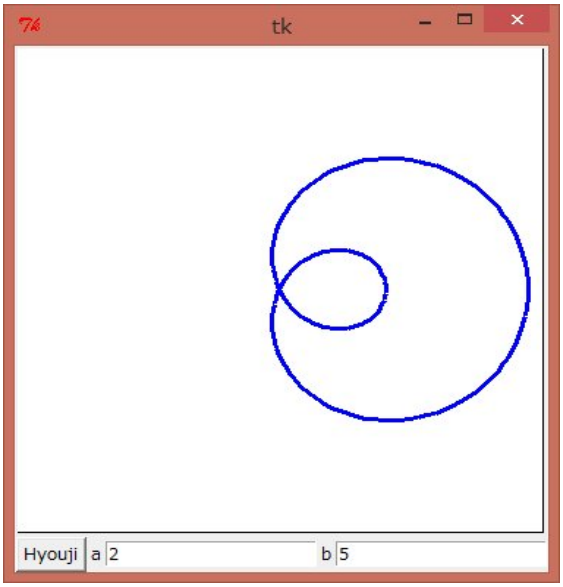
レポート問題：極方程式

$$r = a + b\cos\theta$$

で表される曲線を リマソン という。特に、$a = b$ のとき、極方程式

$$r = a(1 + \cos\theta)$$

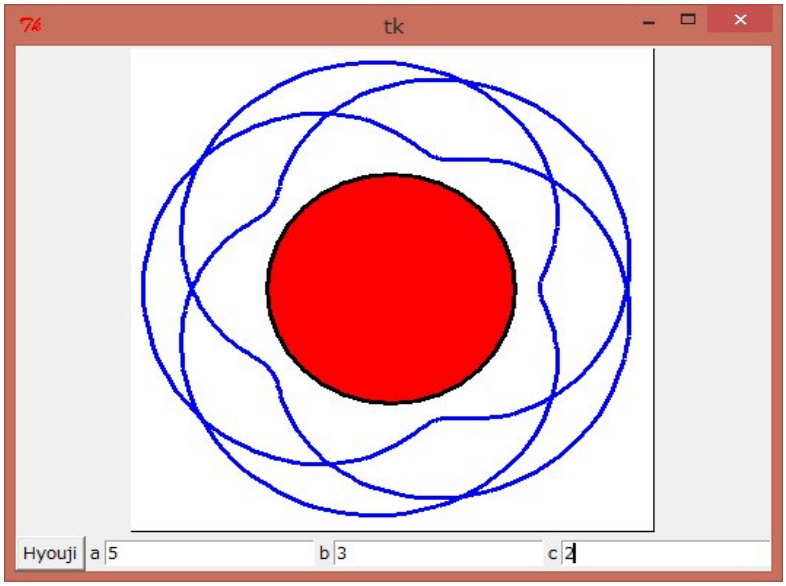
で表される曲線を カージオイド という。リマソンを表示するように、上のプログラムを変更せよ。



レポート問題：

$$x = (a+b)\cos\theta - c\cos\frac{a+b}{b}\theta, \quad y = (a+b)\sin\theta - c\sin\frac{a+b}{b}\theta$$

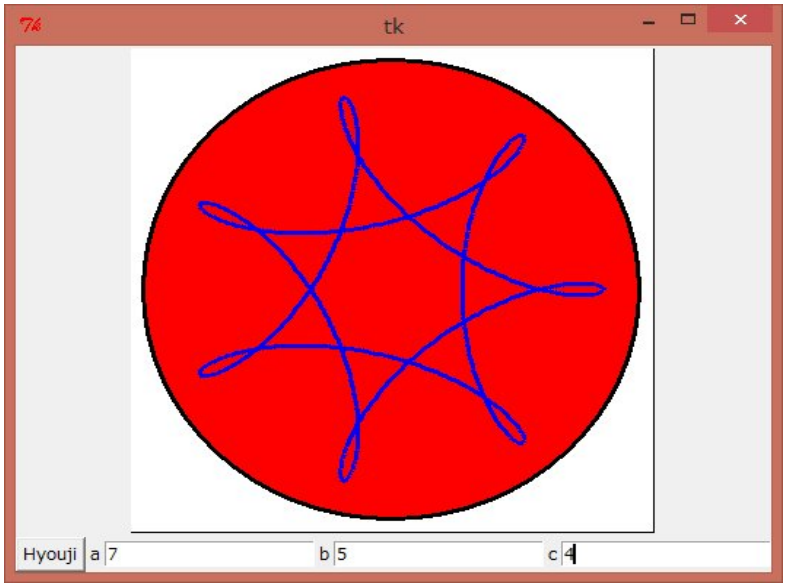
で表される曲線を描くプログラムを作れ。



レポート問題：

$$x = (a-b)\cos\theta + c\cos\frac{a-b}{b}\theta, \quad y = (a-b)\sin\theta - c\sin\frac{a-b}{b}\theta$$

で表される曲線を描くプログラムを作れ。



レポート問題：リサージュ曲線は媒介変数 t を使って

$$x = A\cos(at), \qquad y = B\sin(bt + \delta)$$

で表される曲線です。リサージュ曲線を描くプログラムを作れ。

レポート問題：インボリュート曲線は媒介変数 t を使って

$$x = a(\cos(t) + t\sin(t)), \qquad y = a(\sin(t) - t\cos(t))$$

で表される曲線です。適当な範囲で、インボリュート曲線を描くプログラムを作れ。

おまけ：マンデルブロー集合の描画

マンデルブロー集合は

$z_0 = 0$, $z_{n+1} = z_n^2 + c$ という漸化式で定義される z_n が $n \to \infty$ で発散しない複素数 c の集合です。これを描いてみましょう。

但し、

$|z_n|$ がある定数 K を超えたら発散したと判断します。

n があるループ上限を超えたら発散しなかったと判断します。

これを Turtle Graphics で

```
from turtle import *
import time

def mb(c, K, LOOP):
    z = 0.0 + 0.0*1j
    n = 0
    while (abs(z) < K and n < LOOP):
        z = z**2 + c
        n = n +1
    return n
```

```
def plot(x, y, n, LOOP):
    s = hex(255-n)
    s = s[2:]
    if len(s) == 1:
        s = '0' + s
    cl = '#' + s + s + s
    pencolor(cl)
    pu()
    setpos(100*x, 100*y)
    pd()
    setpos(100*x+1, 100*y+1)

dx, dy = 0.01, 0.01
xmin, xmax = -1.8, 0.6
ymin, ymax = -1.0, 1.0
K = 2.0
LOOP = 255
ht()
start_time = time.time()
x = xmin
while x < xmax:
    y = ymin
    while y < ymax:
        c = x + y*1j
        n = mb(c, K, LOOP)
        plot(x, y, n, LOOP)
        y += dy
    x += dx
end_time = time.time()

print "time = %f" % (end_time-start_time)
```

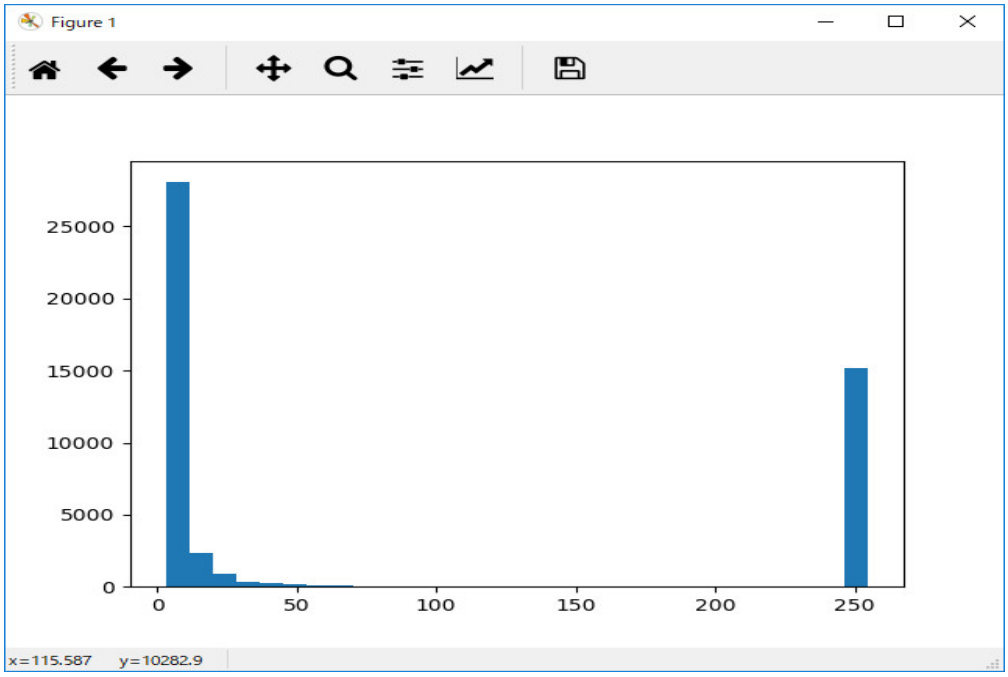
というプログラムで実行しましたが、時間がかかって使い物になりませんでしたし、グラデーションもはっきりしませんでした。ここで、もう一度、この問題に挑戦してみましょう。

ライブラリー pylab は Python に最初からインストールされていませんから、インターネットで `pylab install` を検索し、導入します。英語ですが導入は難しくありません。まずライブラリー pylab を使って、領域外に出るまでの回数の分布を調べてみましょう。

```
import time
import pylab

def mb(x,y):
    c = complex(x, y)
```

```
    z = complex(0.0, 0.0)
    n = 0
    LOOP = 255

    while (abs(z) < 3 and n < LOOP):
        z = z**2 + c
        n = n +1
    return n

start_time = time.time()
dx, dy = 0.01, 0.01
xmin, xmax = -1.8, 0.6
ymin, ymax = -1.0, 1.0
x = pylab.arange(xmin, xmax, dy)
y = pylab.arange(ymin, ymax, dx)
l = []

for i in range(0,len(y)):
    for j in range(0,len(x)):
        l.append(mb(x[j],y[i]))

pylab.hist(l, bins = 30)
pylab.show()
```

を実行します。



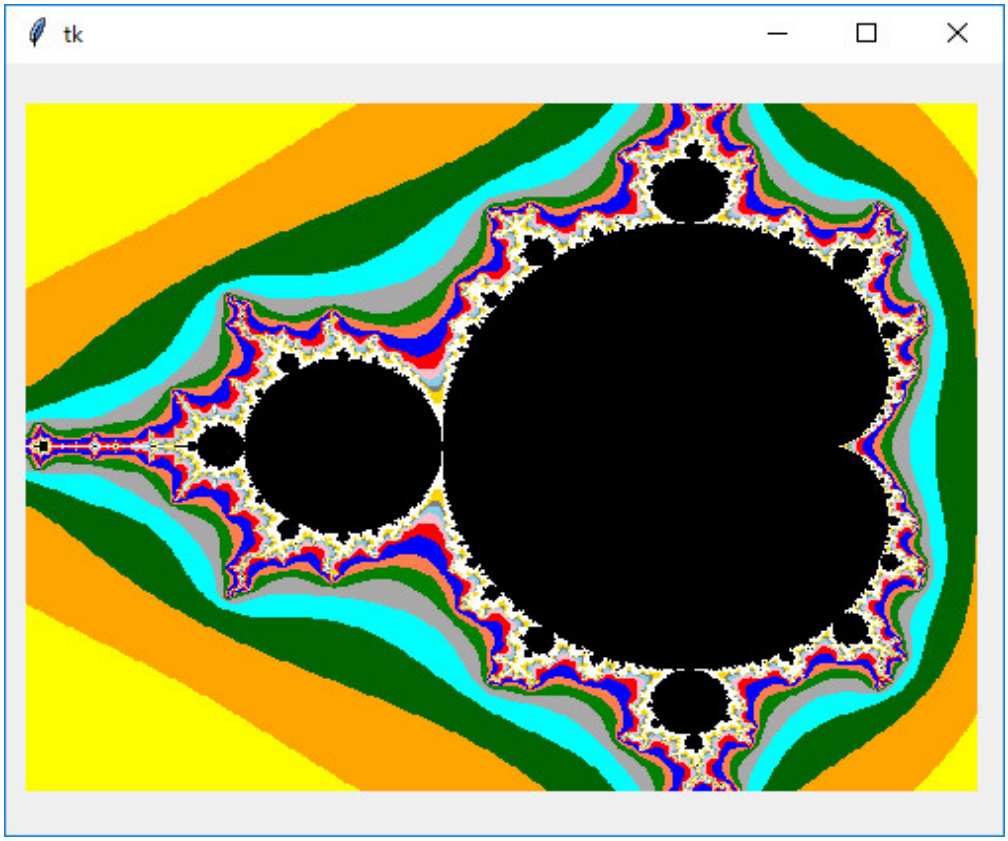
意外な結果です。領域外に出る場合はほとんど１０回以下で出ています。領域外に出るまでの回

数で色分けするためには３０回以下の場合を小さく区別する必要があります。

```
from tkinter import *
import math
import time

root = Tk()
canvas = Canvas(root, width = 500, height=400)
def mb(c, K, LOOP):
    z = 0.0 + 0.0*1j
    n = 0
    while (abs(z) < K and n < LOOP):
        z = z**2 + c
        n = n +1
    return n
def plot(x, y, n):
    gx = 200*x + 370
    gy = -180*y + 200
    if n <= 3:
       canvas.create_line(gx, gy, gx+1, gy+1, fill = 'yellow')
    elif n <= 4:
       canvas.create_line(gx, gy, gx+1, gy+1, fill = 'orange')
    elif n <= 5:
       canvas.create_line(gx, gy, gx+1, gy+1, fill = 'dark green')
    elif n <= 6:
       canvas.create_line(gx, gy, gx+1, gy+1, fill = 'cyan')
    elif n <= 7:
       canvas.create_line(gx, gy, gx+1, gy+1, fill = 'dark gray')
    elif n <= 8:
       canvas.create_line(gx, gy, gx+1, gy+1, fill = 'green')
    elif n < 9:
       canvas.create_line(gx, gy, gx+1, gy+1, fill = 'chocolate')
    elif n < 10:
       canvas.create_line(gx, gy, gx+1, gy+1, fill = 'coral')
    elif n < 12:
       canvas.create_line(gx, gy, gx+1, gy+1, fill = 'blue')
    elif n < 14:
       canvas.create_line(gx, gy, gx+1, gy+1, fill = 'red')
    elif n < 16:
       canvas.create_line(gx, gy, gx+1, gy+1, fill = 'pink')
    elif n < 18:
       canvas.create_line(gx, gy, gx+1, gy+1, fill = 'sky blue')
    elif n < 20:
```

```
        canvas.create_line(gx, gy, gx+1, gy+1, fill = 'gray')
    elif n < 25:
        canvas.create_line(gx, gy, gx+1, gy+1, fill = 'gold')
    elif n < 30:
        canvas.create_line(gx, gy, gx+1, gy+1, fill = 'light yellow')
    elif n < 250:
        canvas.create_line(gx, gy, gx+1, gy+1, fill = 'white')
    else:
        canvas.create_line(gx, gy, gx+1, gy+1, fill = 'black')
dx, dy = 0.005, 0.005
xmin, xmax = -1.8, 0.6
ymin, ymax = -1.0, 1.0
K = 3.0
LOOP = 255
start_time = time.time()
x = xmin
while x < xmax:
    y = ymin
    while y < ymax:
        c = x + y*1j
        n = mb(c, K, LOOP)
        plot(x, y, n)
        y += dy
    x += dx
end_time = time.time()
print( "time = %f" % (end_time-start_time) )
canvas.pack()
root.mainloop()
```

を実行します。



を得ました。色分け等自分で工夫してみてください。時間は１４秒ぐらいです。

実は上のプログラムは適当に作っています。きちんと計算してプログラムを作ると次のようになります。まずキャンバスの大きさを変更しています。さらにマンデルブロー集合の全体を描いた後、マウスで左ボタンを押しながら範囲を指定するとその部分を拡大して再描画するようにします。そのために、点を塗る色の指定方法も n の値に応じてサイクリックに指定しています。

```
    #gx = 200*x + 370
    #gy = -180*y + 200
```

のように # のついている行はコメントです。もとのプログラムではこのようになっていたことを残しています。

プログラムは次のようになります。

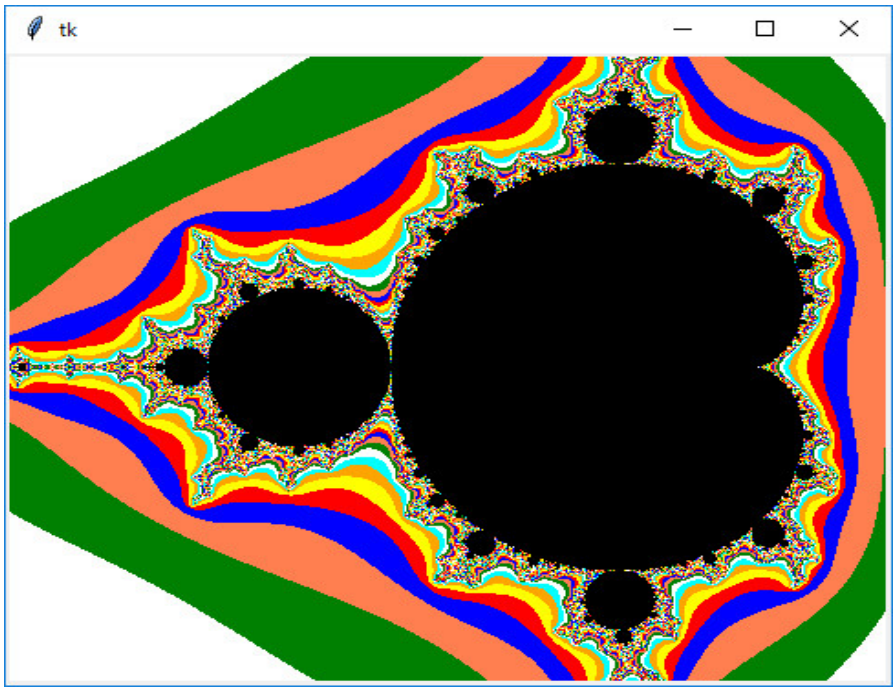
```
from tkinter import *
import math
import time

root = Tk()
canvas = Canvas(root, width = 480, height=400)
sx , sy = 0.0, 0.0
def setsource(event):
    global sx, sy
```

```
    sx = (xmax-xmin)/480.0*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)/480.0*event.x + xmin
    ty = ymax - (ymax-ymin)/400.0*event.y
    gsx = 480/(xmax-xmin)*(sx-xmin)
    gsy = 400/(ymax-ymin)*(ymax-sy)
    gtx = 480/(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)/480.0*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)/480.0
    dy = (ymax-ymin)/400.0
    x = xmin
    while x < xmax:
        y = ymin
        while y < ymax:
            c = x + y*1j
            n = mb(c, K, LOOP)
            plot(x, y, n, LOOP)
            y += dy
        x += dx

canvas.bind("<ButtonRelease-1>", redraw)

def mb(c, K, LOOP):
    z = 0.0 + 0.0*1j
    n = 0
```

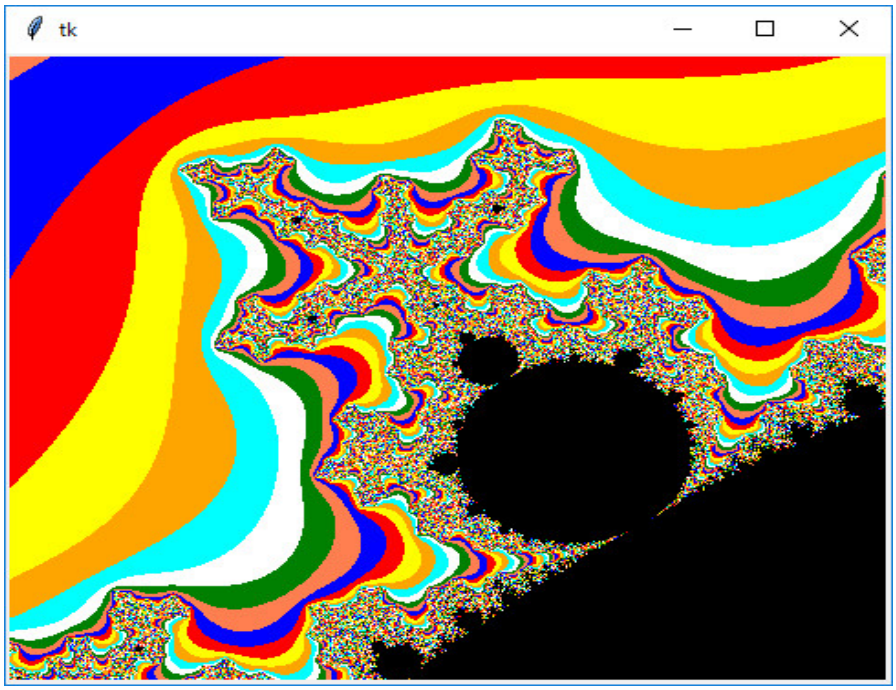
```
    while (abs(z) < K and n < LOOP):
        z = z**2 + c
        n = n +1
    return n
def plot(x, y, n, LOOP):
    #gx = 200*x + 370
    #gy = -180*y + 200
    gx = 480/(xmax-xmin)*(x-xmin)
    gy = 400/(ymax-ymin)*(ymax-y)
    if n == LOOP:
       canvas.create_line(gx, gy, gx+1, gy+1, fill = 'black')
    elif n % 8 == 0:
       canvas.create_line(gx, gy, gx+1, gy+1, fill = 'yellow')
    elif n % 8 == 1:
       canvas.create_line(gx, gy, gx+1, gy+1, fill = 'orange')
    elif n % 8 == 2:
       canvas.create_line(gx, gy, gx+1, gy+1, fill = 'cyan')
    elif n % 8 == 3:
       canvas.create_line(gx, gy, gx+1, gy+1, fill = 'white')
    elif n % 8 == 4:
       canvas.create_line(gx, gy, gx+1, gy+1, fill = 'green')
    elif n % 8 == 5:
       canvas.create_line(gx, gy, gx+1, gy+1, fill = 'coral')
    elif n % 8 == 6:
       canvas.create_line(gx, gy, gx+1, gy+1, fill = 'blue')
    elif n % 8 == 7:
       canvas.create_line(gx, gy, gx+1, gy+1, fill = 'red')
dx, dy = 0.005, 0.005
xmin, xmax = -1.8, 0.6
ymin, ymax = -1.0, 1.0

K = 3.0
LOOP = 255
#start_time = time.time()
x = xmin
while x < xmax:
    y = ymin
    while y < ymax:
        c = x + y*1j
        n = mb(c, K, LOOP)
        plot(x, y, n, LOOP)
        y += dy
    x += dx
```

```
#end_time = time.time()
#print( "time = %f" % (end_time-start_time) )
canvas.pack()
root.mainloop()
```

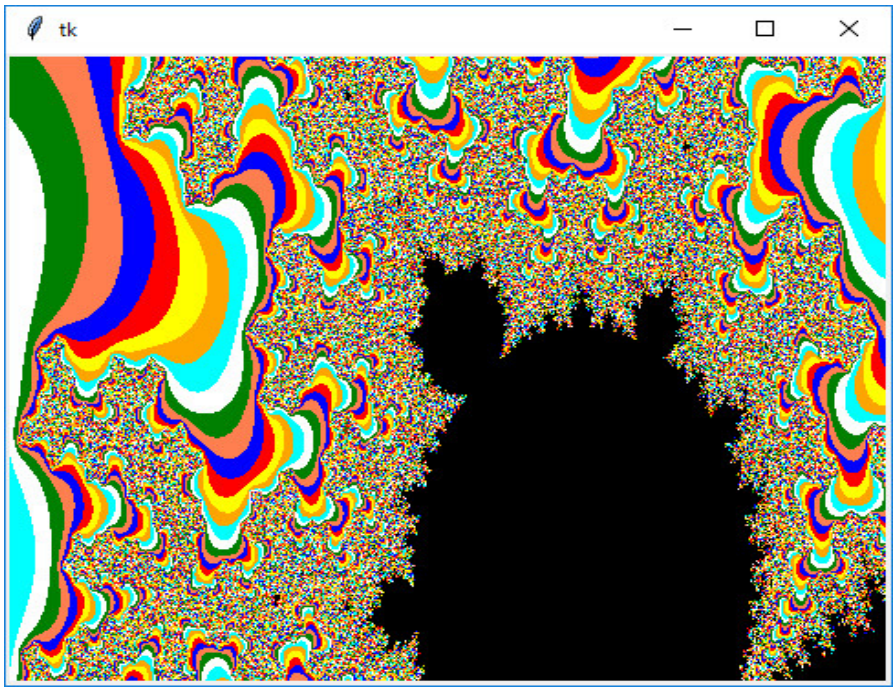
実行すると



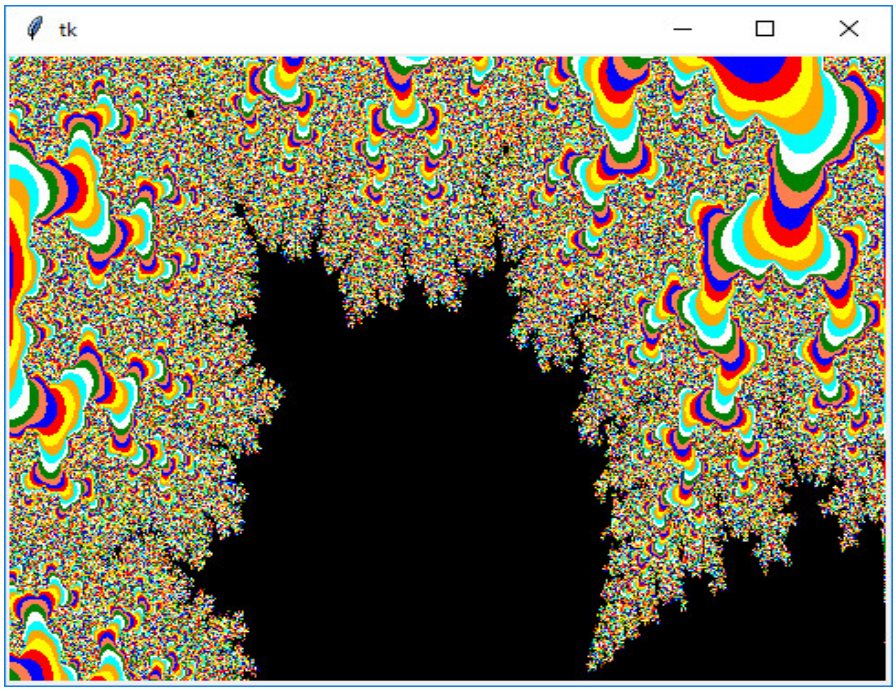
です。マウスでドラッグし、範囲を指定する（どこを指定したかの画像はコピーできませんでしたが）と



のように、指定した部分を拡大して描画します。
　さらに、マウスでドラッグし、範囲を指定すると



のように、指定した部分を拡大して描画します。
　さらに、マウスでドラッグし、範囲を指定すると



のように、指定した部分を拡大して描画します。但し、この図のように範囲が小さくなり黒い部分が多いとほとんどの点で LOOP 回ループを繰り返しますので、描き終わるのに凄く時間がかかり

ます。このようなプログラムは **Python3.6** でも実行できますが、まだ **Python2.7** で実行した方がいいみたいです。

```
from tkinter import *
```

を

```
from Tkinter import *
```

と変えるだけです。

```
sx , sy = 0.0, 0.0
def setsource(event):
    global sx, sy
    sx = (xmax-xmin)/480.0*event.x + xmin
    sy = ymax - (ymax-ymin)/400.0*event.y
canvas.bind("<Button-1>", setsource)
```

の部分で、キャンバス上で左マウスボタンが押された時の処理をプログラミングしています。global 変数 sx と sy に左マウスボタンが押された時の座標をセットしています。

```
tx, ty = 0.0, 0.0
def drawline(event):
    global sx, sy, tx, ty
    tx = (xmax-xmin)/480.0*event.x + xmin
    ty = ymax - (ymax-ymin)/400.0*event.y
    gsx = 480/(xmax-xmin)*(sx-xmin)
    gsy = 400/(ymax-ymin)*(ymax-sy)
    gtx = 480/(xmax-xmin)*(tx-xmin)
    gty = 400/(ymax-ymin)*(ymax-ty)
    canvas.create_rectangle(gsx, gsy, gtx, gty)
canvas.bind("<B1-Motion>", drawline)
```

の部分で、キャンバス上で左ボタンが押しながらマウスを動かした時の処理をプログラミングしています。global 変数 tx と ty にマウスの座標をセットして、(sx, sy) と (tx, ty) を角の座標とする長方形を描いています。これは一定の時間ごとに呼ばれます。

```
def redraw(event):
    global sx, sy, tx, ty
    global xmax, xmin, ymax, ymin
    global dx, dy
    tx = (xmax-xmin)/480.0*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)/480.0
    dy = (ymax-ymin)/400.0
    x = xmin
    while x < xmax:
        y = ymin
        while y < ymax:
            c = x + y*1j
            n = mb(c, K, LOOP)
            plot(x, y, n, LOOP)
            y += dy
        x += dx
canvas.bind("<ButtonRelease-1>", redraw)
```

の部分で、キャンバス上で左ボタンを押してそのボタンを放した時の処理をプログラミングしています。キャンバス上で左マウスボタンが押された時の座標とそのボタンを放した時の座標をもとに描画すべき範囲を計算し、刻み幅 dx と dy を再計算し、画面を描画しています。この部分の処理に時間がかかります。これが時間がかかりすぎると考えるなら、コンパイラの C++ を勉強してください。Python のようなインタープリターは複雑な計算をさせるとどうしても直接機械語に翻訳して実行するコンパイラに比べると時間がかかります。一般的には、上の例題のような何度も使う計算量が多く時間のかかるプログラムは、 Python は手軽にプログラミングできるので、プロトタイプを Python で作り、上手くいったら、C++　で実用版を作ればいいです。

マンデルブロー集合と似たものにジュリア集合があります。ジュリア集合は、複素 2 次関数 $f(z) = z^2 + C$ を定義し、定数 $C = a + bi$ を定め、複素平面上の各点が漸化式 $z_{n+1} = f(z_n)$ によって収束するか、発散するかを判定して、収束領域と発散領域を表示したものである。発散領域については、発散の速さで表示方法を変える。ここでも、プログラミングするときは次のようにします。

$|z_n|$ がある定数 K を超えたら発散したと判断します。

n があるループ上限を超えたら発散しなかったと判断します。

プログラムの例は次のようになります。

```
from tkinter import *
import math

root = Tk()
f0 = Frame(root)
f1 = Frame(root)
canvas = Canvas(f0, width = 400, height=400)
a = -0.3
b = -0.63
K = 3.0
LOOP = 255
C = complex(0.0, 0.0)
xmin, xmax = -2.0, 2.0
```

```
ymin, ymax = -2.0, 2.0
def paint():
    global C, xmin, xmax, ymin, ymax
    a = float(content1.get())
    b = float(content2.get())
    C = complex(a, b)
    dx, dy = 0.01, 0.01
    xmin, xmax = -2.0, 2.0
    ymin, ymax = -2.0, 2.0

    x = xmin
    while x < xmax:
        y = ymin
        while y < ymax:
            z = complex(x, y)
            n = julia(z, K, LOOP)
            plot(x, y, n, LOOP)
            y += dy
        x += dx

button = Button(f1, text='PAINT', command=paint).pack(side=LEFT)
l1 = Label(f1, text='a').pack(side=LEFT)
content1 = StringVar()
t1 = Entry(f1, textvariable=content1).pack(side=LEFT)
l2 = Label(f1, text='b').pack(side=LEFT)
content2 = StringVar()
t2 = Entry(f1, textvariable=content2).pack(side=LEFT)
f0.pack()
f1.pack()

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)
```

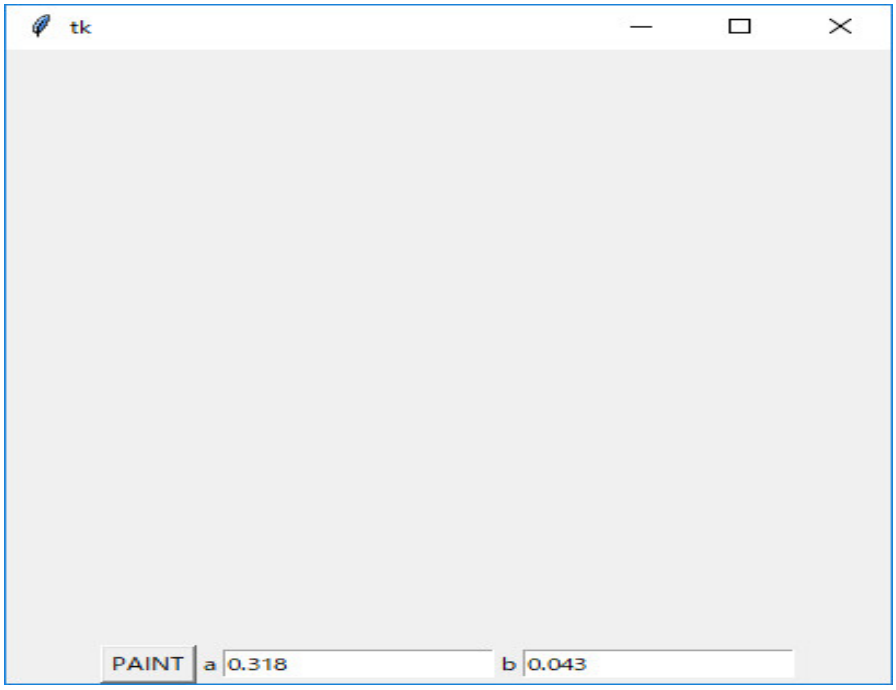
```
    gtx = 400/(xmax-xmin)*(tx-xmin)
    gty = 400/(ymax-ymin)*(ymax-ty)
    canvas.create_rectangle(gsx, gsy, gtx, gty)
canvas.bind("<B1-Motion>", drawline)
def redraw(event):
    global sx, sy, tx, ty
    global xmax, xmin, ymax, ymin
    global dx, dy
    tx = (xmax-xmin)/400.0*event.x + xmin
    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)
            n = julia(z, K, LOOP)
            plot(x, y, n, LOOP)
            y += dy
        x += dx

canvas.bind("<ButtonRelease-1>", redraw)

def julia(z, K, LOOP):
    global C
    n = 0
    while (abs(z) < K and n < LOOP):
        z = z**2 + C
        n = n +1
    return n
def plot(x, y, n, LOOP):
    gx = 400/(xmax-xmin)*(x-xmin)
    gy = 400/(ymax-ymin)*(ymax-y)
    if n == LOOP:
       canvas.create_line(gx, gy, gx+1, gy+1, fill = 'black')
    elif n % 8 == 0:
```

```
        canvas.create_line(gx, gy, gx+1, gy+1, fill = 'yellow')
    elif n % 8 == 1:
        canvas.create_line(gx, gy, gx+1, gy+1, fill = 'orange')
    elif n % 8 == 2:
        canvas.create_line(gx, gy, gx+1, gy+1, fill = 'cyan')
    elif n % 8 == 3:
        canvas.create_line(gx, gy, gx+1, gy+1, fill = 'white')
    elif n % 8 == 4:
        canvas.create_line(gx, gy, gx+1, gy+1, fill = 'green')
    elif n % 8 == 5:
        canvas.create_line(gx, gy, gx+1, gy+1, fill = 'coral')
    elif n % 8 == 6:
        canvas.create_line(gx, gy, gx+1, gy+1, fill = 'blue')
    elif n % 8 == 7:
        canvas.create_line(gx, gy, gx+1, gy+1, fill = 'red')

content1.set('%.3f' % (0.318))
content2.set('%.3f' % (0.043))
canvas.pack()
root.mainloop()
```

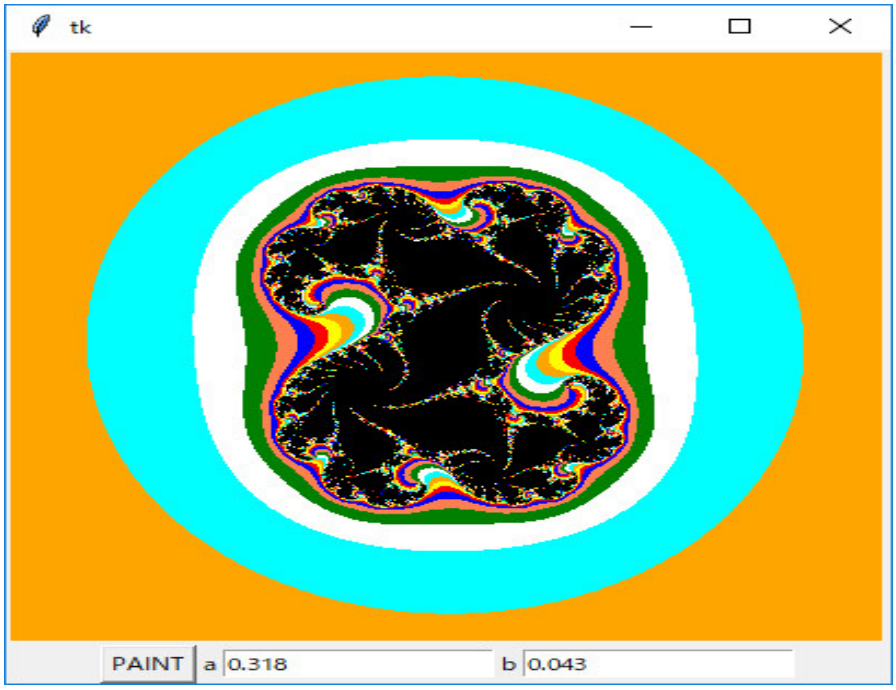
ここで

```
content1.set('%.3f' % (0.318))
content2.set('%.3f' % (0.043))
```

で、テキストボックスに表示する値をセットしています。
実行すると

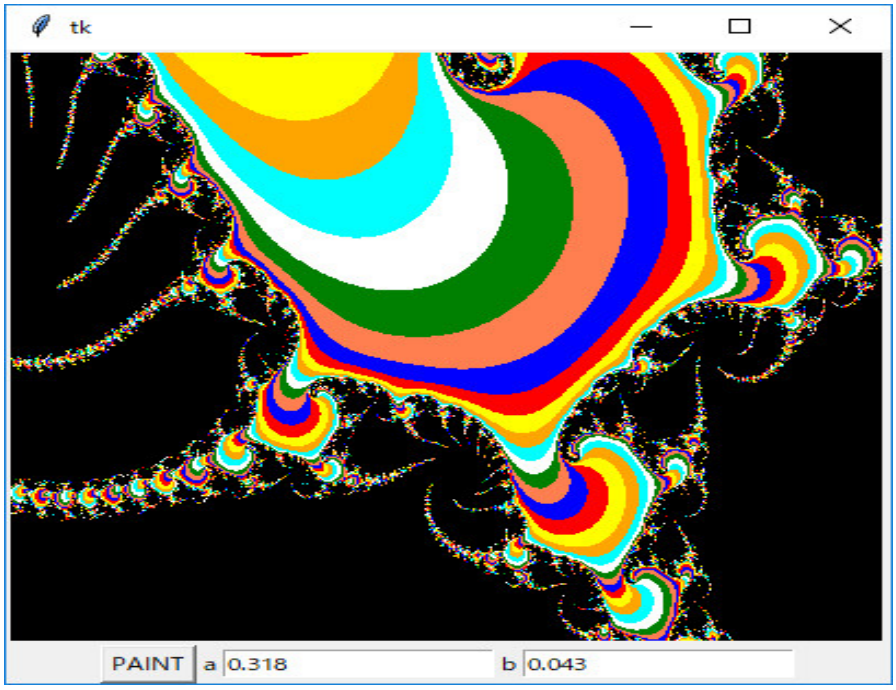


となります。複素数の実部と虚部をセットして「PAINT」のボタンをクリックします。そのまま「PAINT」のボタンをクリックすると



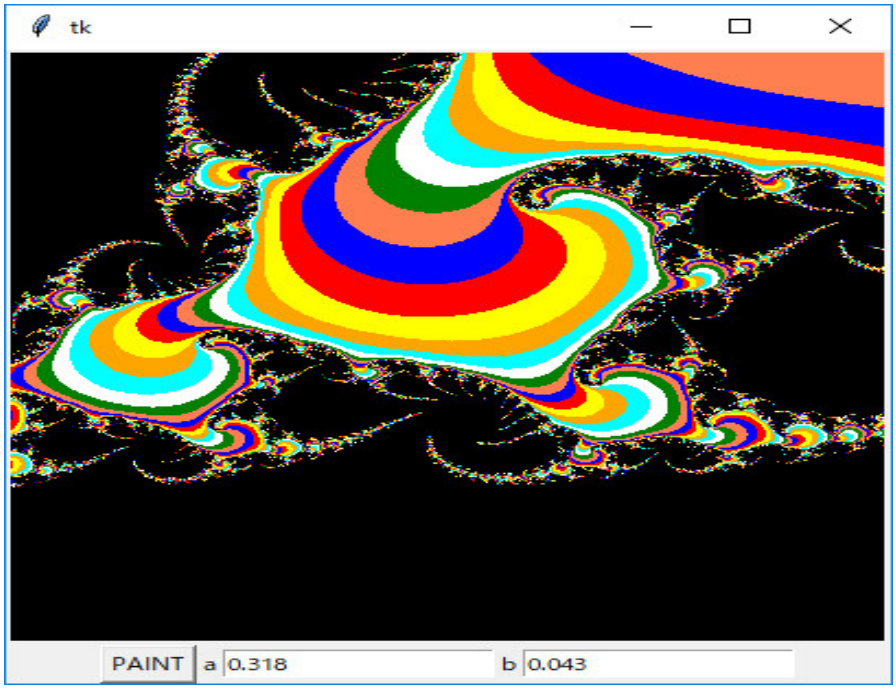
となります。

マウスでドラッグし、範囲を指定する（どこを指定したかの画像はコピーできませんでしたが）と



のように、指定した部分を拡大して描画します。

さらに、マウスでドラッグし、右上の範囲を指定すると



のように、指定した部分を拡大して描画します。このプログラムも **Python3.6** でも実行できますが、まだ **Python2.7** で実行した方がいいみたいです。

指定する複素数によってはつまらない図になります。インターネットにはジュリア集合の美しい図がたくさん載っていますが、どのような複素数をセットすれば、どのような図を描画するか私は

知りません。マンデルブロー集合と関係があるみたいです。また、上の $f(z) = z^2 + C$ を別の関数にして図を描画している人もいます。自分で調べていろいろ試してみてください。

最後に、常微分方程式の解を図示してみましょう。ここでは簡単なオイラー法を使います。オイラー法での数値計算は常微分方程式

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

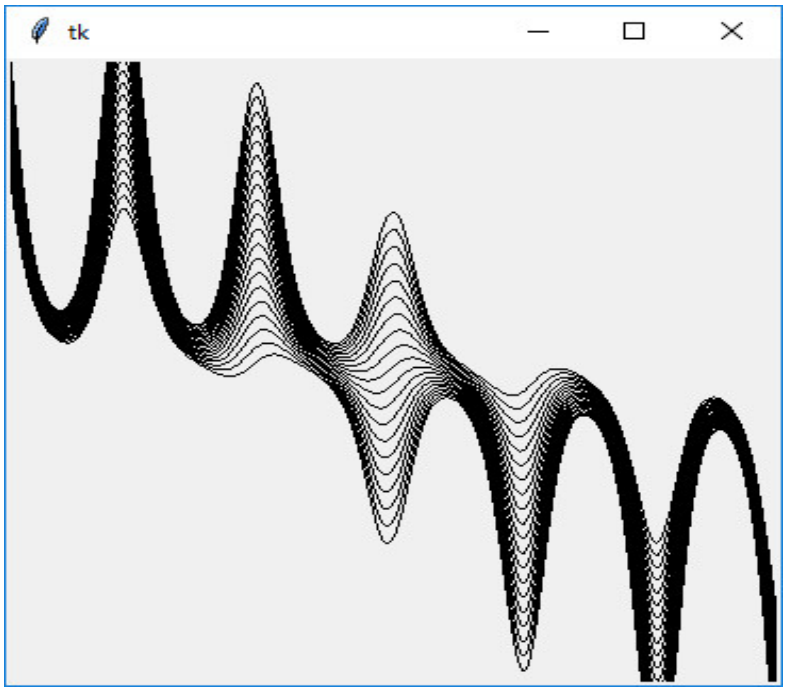
に対して、漸化式

$$\begin{cases} x_{i+1} = x_i + \Delta x \\ y_{i+1} = y_i + f(x_i, y_i)\Delta x \end{cases}$$

を使えば、初期値 (x_0, y_0) が決まれば、順次 $(x_1, y_1), (x_2, y_2), (x_3, y_3), \cdots$ が決まるので、それらを結んで解曲線を描けばいいです。プログラムは単純で

```
from tkinter import *
from math import *

def f(x,y):
    return cos(x)-y*sin(x)

root = Tk()
canvas = Canvas(root, width = 360, height=360)
canvas.pack()

sy = -10
for sy in range(-10,10):
    x0 = 0
    y0 = sy
    deltax = 0.1
    while 10*x0 < 360:
        x = x0 + deltax
        y = y0 + f(x0, y0)*deltax
        canvas.create_line(180+10*x0, 180-10*y0,180+10*x, 180-10*y)
        x0 = x
        y0 = y
    x0 = 0
    y0 = sy
    while 10*x0 > -360:
        x = x0 - deltax
        y = y0 - f(x0, y0)*deltax
        canvas.create_line(180+10*x0, 180-10*y0,180+10*x, 180-10*y)
        x0 = x
        y0 = y

root.mainloop()
```

のようなもので、



のような図を描いてくれます。初期値を $(0,-10),(0,-9),(0,-8),\cdots,(0,9)$ と変化しながら、常微分方程式

$$\frac{dy}{dx}=\cos(x)-y\sin(x)$$

の解曲線を描いています。

関数 $f(x,y)$ を好きな関数に変えて、色々確かめてみれば良いです。**Python 2.7** では

```
from Tkinter import *
```

と **tkinter** を大文字にすればいいです。

もっと正確な解曲線を描くには、ルンゲ・クッタ法があります。古典的ルンゲ・クッタ法は常微分方程式

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

に対して、漸化式

$$\begin{cases} x_{i+1}=x_i+\Delta x \\ y_{i+1}=y_i+\dfrac{\Delta x}{6}(k_1+2k_2+2k_3+k_4) \end{cases}$$

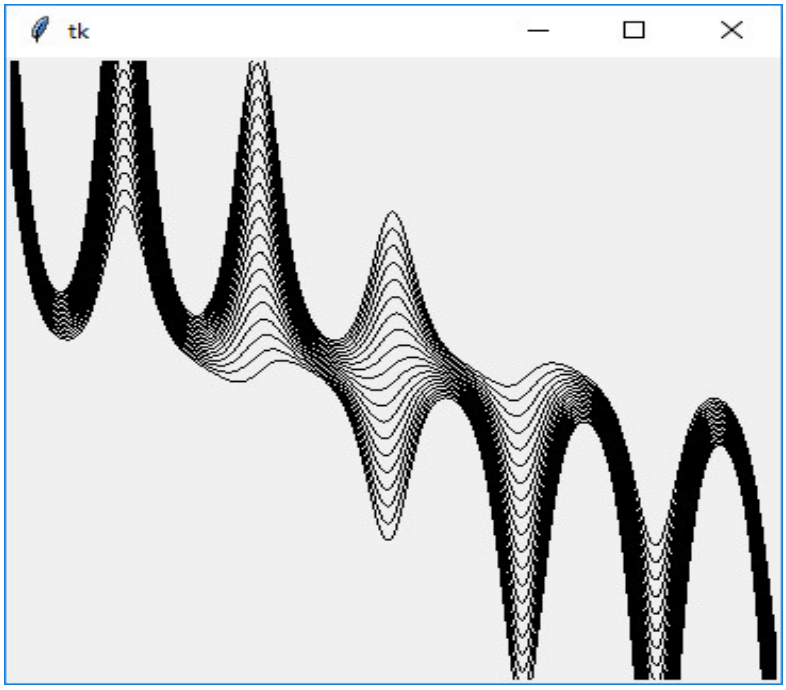
ここで

$$\begin{aligned} k_1&=f(x_i,y_i)\\ k_2&=f(x_i+\frac{\Delta x}{2},y_i+\frac{\Delta x}{2}k_1)\\ k_3&=f(x_i+\frac{\Delta x}{2},y_i+\frac{\Delta x}{2}k_2)\\ k_4&=f(x_i+\Delta x,y_i+\Delta x k_3) \end{aligned}$$

を使えば、初期値 (x_0, y_0) が決まれば、順次 $(x_1, y_1), (x_2, y_2), (x_3, y_3), \cdots$ が決まるので、それらを結んで解曲線を描けばいいです。

```
from tkinter import *
from math import *

def f(x,y):
    return cos(x)-y*sin(x)

root = Tk()
canvas = Canvas(root, width = 360, height=360)
canvas.pack()

sy = -10
for sy in range(-10,10):
    x0 = 0
    y0 = sy
    deltax = 0.1
    while 10*x0 < 360:
        x = x0 + deltax
        k1 = f(x0, y0)
        k2 = f(x0+deltax/2, y0+deltax*k1/2)
        k3 = f(x0+deltax/2, y0+deltax*k2/2)
        k4 = f(x0+deltax, y0+deltax*k3)
        y = y0 + (k1+2*k2+2*k3+k4)*deltax/6
        canvas.create_line(180+10*x0, 180-10*y0,180+10*x, 180-10*y)
        x0 = x
        y0 = y
    x0 = 0
    y0 = sy
    while 10*x0 > -360:
        x = x0 - deltax
        k1 = f(x0, y0)
        k2 = f(x0-deltax/2, y0-deltax*k1/2)
        k3 = f(x0-deltax/2, y0-deltax*k2/2)
        k4 = f(x0-deltax, y0-deltax*k3)
        y = y0 - (k1+2*k2+2*k3+k4)*deltax/6
        canvas.create_line(180+10*x0, 180-10*y0,180+10*x, 180-10*y)
        x0 = x
        y0 = y

root.mainloop()
```

のプログラムで



の図となります。

２階常微分方程式

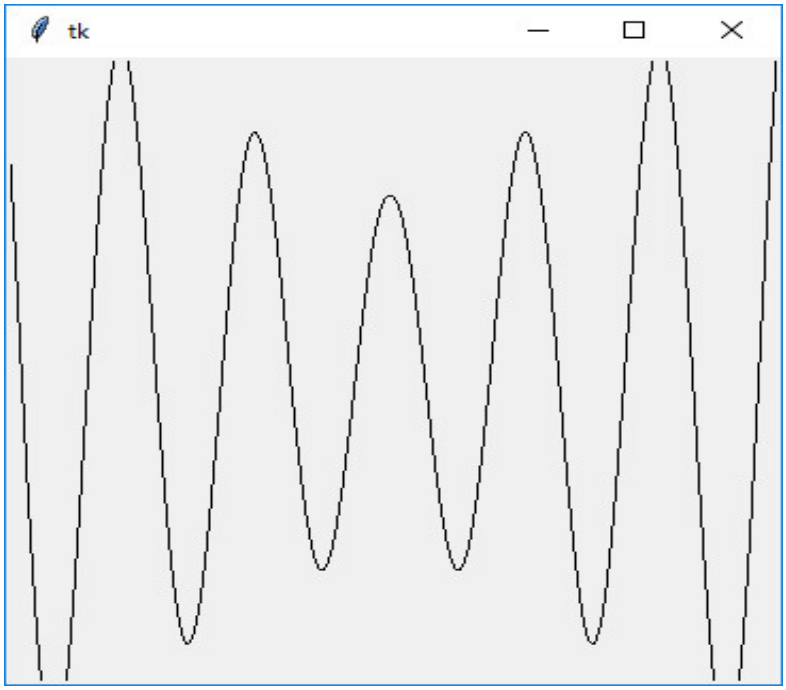
$$\frac{d^2y}{dt^2} = -y$$

の解曲線はオイラー法では

```
from tkinter import *
from math import *

root = Tk()
canvas = Canvas(root, width = 360, height=360)
canvas.pack()

t0 = 0
u0 = 0
y0 = 10
delta_t = 0.1
while 10*t0 < 360:
    t = t0 + delta_t
    u = u0 - y0 * delta_t
    y = y0 + u0 * delta_t
    canvas.create_line(180+10*t0, 180-10*y0,180+10*t, 180-10*y)
    t0 = t
    u0 = u
    y0 = y
t0 = 0
```

```
u0 = 0
y0 = 10
while 10*t0 > -360:
    t = t0 - delta_t
    u = u0 + y0 * delta_t
    y = y0 - u0 * delta_t
    canvas.create_line(180+10*t0, 180-10*y0,180+10*t, 180-10*y)
    t0 = t
    u0 = u
    y0 = y

root.mainloop()
```

のプログラムで与えられ、初期値 $(t_0 = 0, u_0 = 0, y_0 = 10)$ の解曲線は



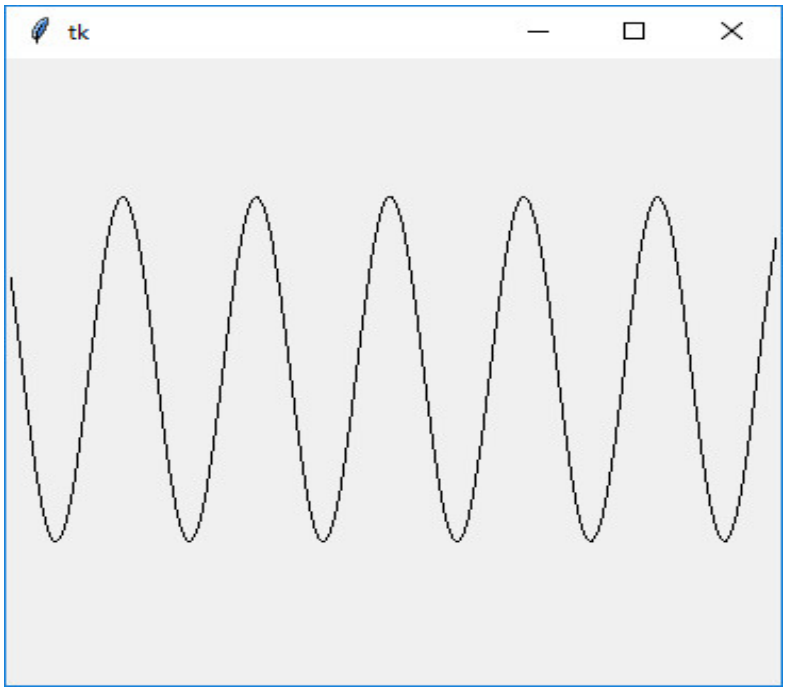
のような図を描きます。一般解は $y = A\cos(t) + B\sin(t)$ ですから、オイラー法では誤差が随分大きいです。もっと正確な解曲線が欲しい時には、ルンゲ・クッタ法を使います。ルンゲ・クッタ法のプログラムは

```
from tkinter import *
from math import *

root = Tk()
canvas = Canvas(root, width = 360, height=360)
canvas.pack()

t0 = 0
```

```
u0 = 0
y0 = 10
delta_t = 0.1
while 10*t0 < 360:
    k1 = u0*delta_t
    l1 = -y0*delta_t
    k2 = (u0+l1/2.0)*delta_t
    l2 = -(y0+k1/2.0)*delta_t
    k3 = (u0+l2/2.0)*delta_t
    l3 = -(y0+k2/2.0)*delta_t
    k4 = (u0+l3)*delta_t
    l4 = -(y0+k3)*delta_t
    k = (k1+2.0*(k2+k3)+k4)/6.0
    l = (l1+2.0*(l2+l3)+l4)/6.0
    t = t0 + delta_t
    u = u0 + l
    y = y0 + k
    canvas.create_line(180+10*t0, 180-10*y0,180+10*t, 180-10*y)
    t0 = t
    u0 = u
    y0 = y
t0 = 0
u0 = 0
y0 = 10
delta_t = 0.1
while 10*t0 > -360:
    k1 = -u0*delta_t
    l1 = y0*delta_t
    k2 = -(u0+l1/2.0)*delta_t
    l2 = (y0+k1/2.0)*delta_t
    k3 = -(u0+l2/2.0)*delta_t
    l3 = (y0+k2/2.0)*delta_t
    k4 = -(u0+l3)*delta_t
    l4 = (y0+k3)*delta_t
    k = (k1+2.0*(k2+k3)+k4)/6.0
    l = (l1+2.0*(l2+l3)+l4)/6.0
    t = t0 - delta_t
    u = u0 + l
    y = y0 + k
    canvas.create_line(180+10*t0, 180-10*y0,180+10*t, 180-10*y)
    t0 = t
    u0 = u
    y0 = y
```

```
root.mainloop()
```

のようになり、結果は



の図となります。三角関数のグラフらしく見えます。詳しく知りたい人は加納先生に聞いてください。

ライブラリー **tkinter** を使ったプログラミングの例として、「数独のヒントを表示するプログラム」と「トランプゲームのプログラム」と「電子ピアノのプログラム」を情報数学のホームページに載せてあるのでそれも参照してください。

ライブラリー **pylab** の使い方は、**MIT** （マサチューセッツ工科大学）の初心者教育のためのテキスト： **John V. Guttag** 著 「**Introduction to Computation and Programming Using Python**」を読むといいです。流石世界最高クラスの大学の教科書だけあって、素晴らしい本です。翻訳もありますが原書のほうが１０００円くらい安いですし、やさしい英語で書かれているので是非原書で読んでください。英語の勉強にもなります。私が説明できなかったオブジェクト指向のクラスの作り方や例外処理を含む **Python** の基本的な事柄を簡潔にしかも漏らさず説明し、基本的なアルゴリズム等を説明した後、後半はライブラリー **pylab** を使った **Python** による統計処理を、最後は最近注目されている機械学習まで簡潔に例題付きで説明しています。私が最近読んだ本の中で最高の本でした。ここまで **Python** を勉強してきましたから、もう独学でこの本を読む力がついているはずです。この授業の次に学ぶべきこととして、コンピュータサイエンスの基礎知識を身に着けるために、是非この本を読んでみてください。