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

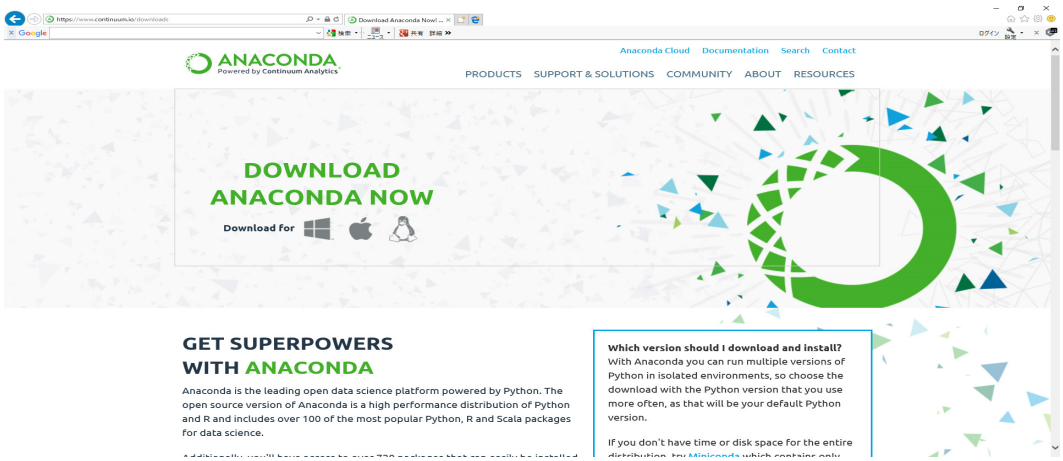
1　Python によるタートルグラフィックス

ここでタートルグラフィックスを学びます。タートルグラフィックスはシーモア・パパートたちが子供の教育用に開発したプログラミング言語 LOGO で導入されたものです。ここでは Python という最近注目を浴びているフリーのプログラミング言語（オブジェクト指向言語）を使ってみます。

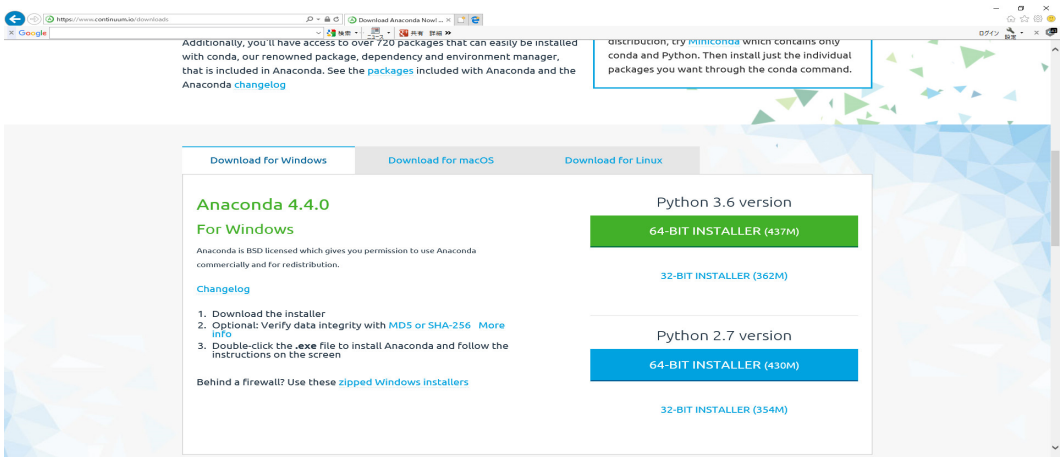
まず Python をインストールして、Python を使えるようにしましょう。ここでは Anaconda を使って Python をインストールします。Anaconda は Python 本体に加えて、よく使われるパッケージを一括してインストールできるようにしたものです。

`https://www.continuum.io/downloads`

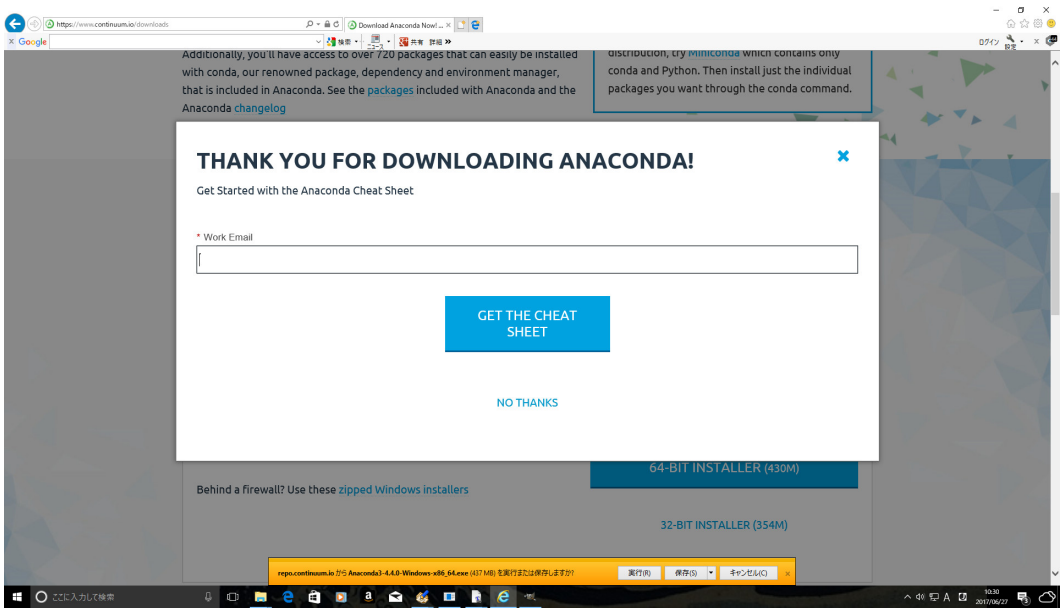
にアクセスします。



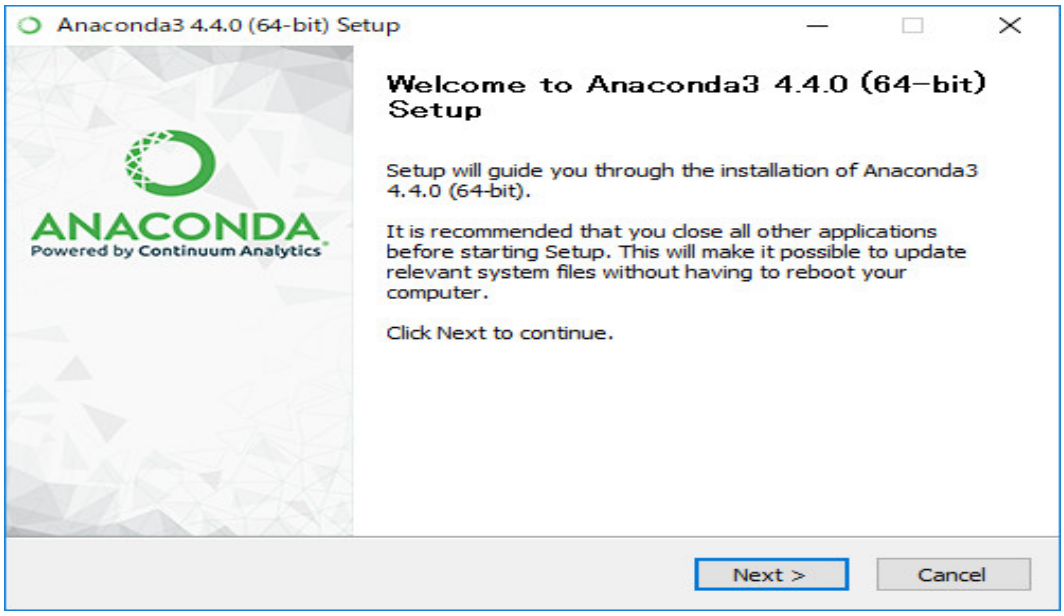
下の方の



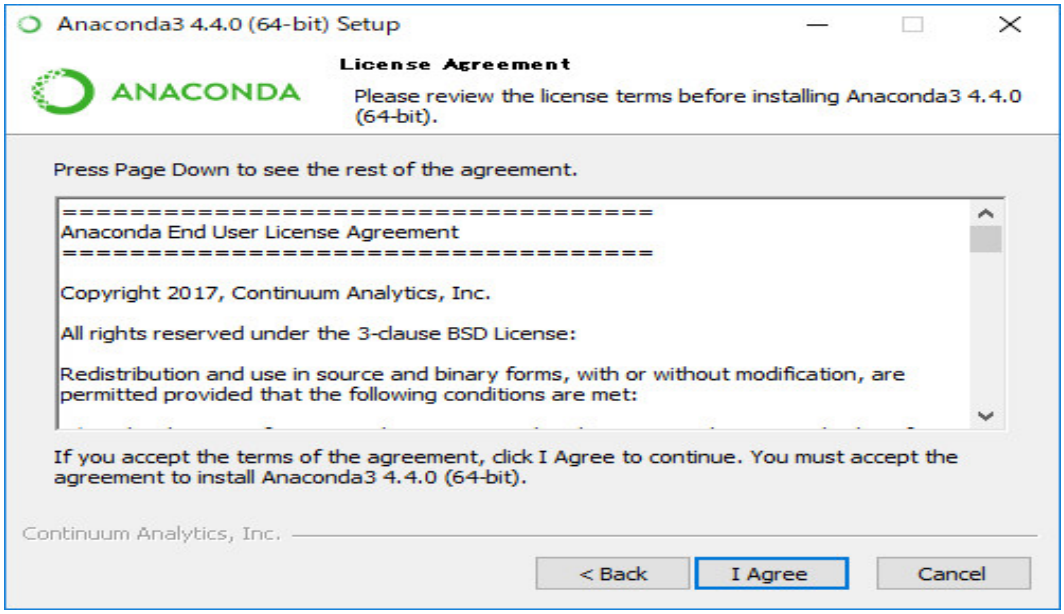
Python 3.6 version をクリックします。現在 Python2 と Python3 が利用できますが、Python 2.7 と Python 3.6 は上位互換性はなく、Python2 は新たな改善はなされないことが決まっているので、これからは Python3 を使って行くのが良いです。



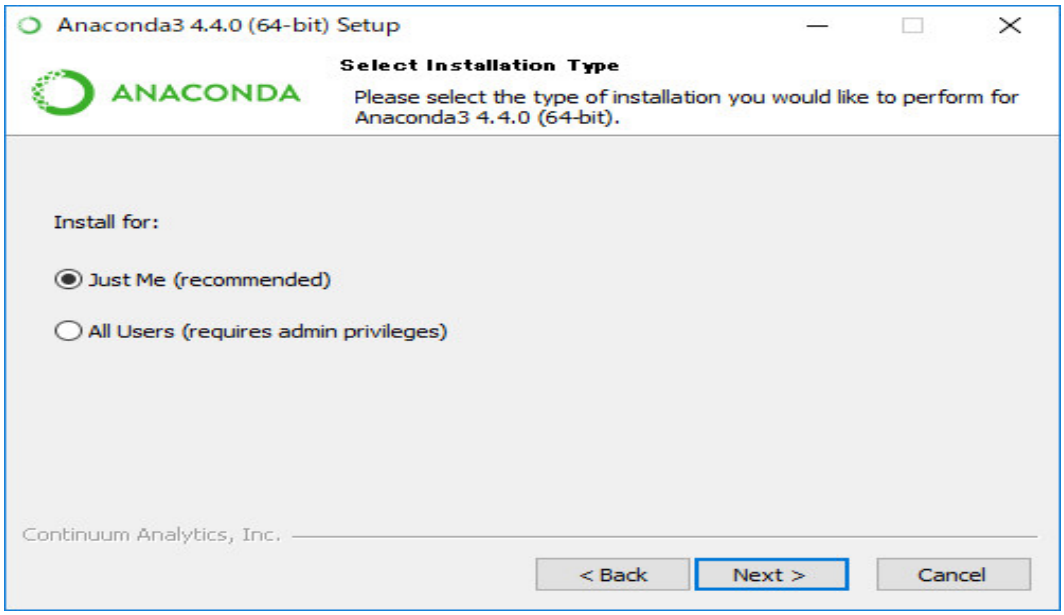
これは無視して、閉じて良いです。「実行」をクリックします。



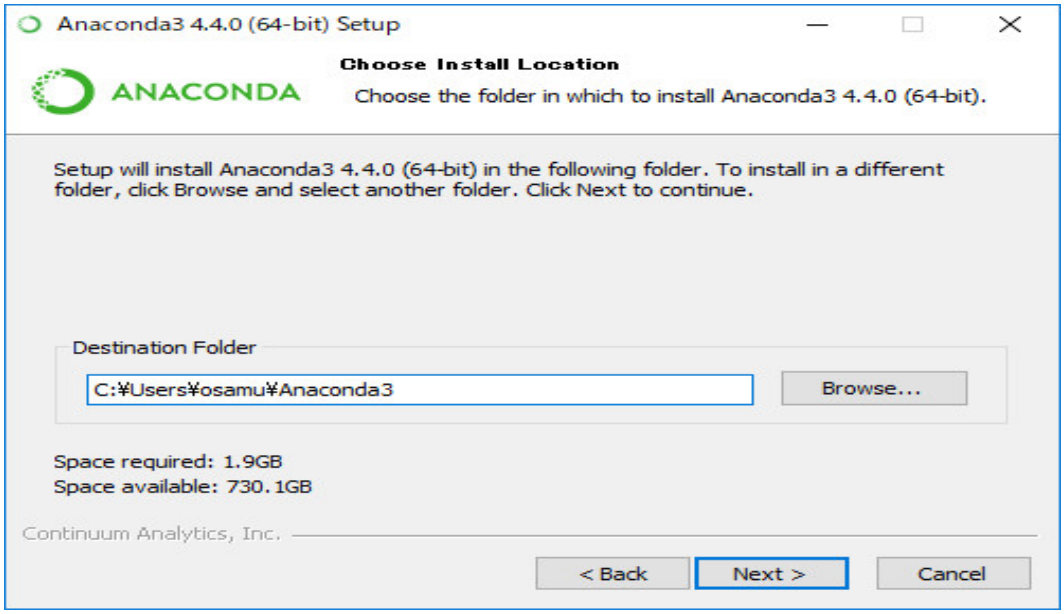
「Next」をクリックします。



「I Agree」をクリックします。



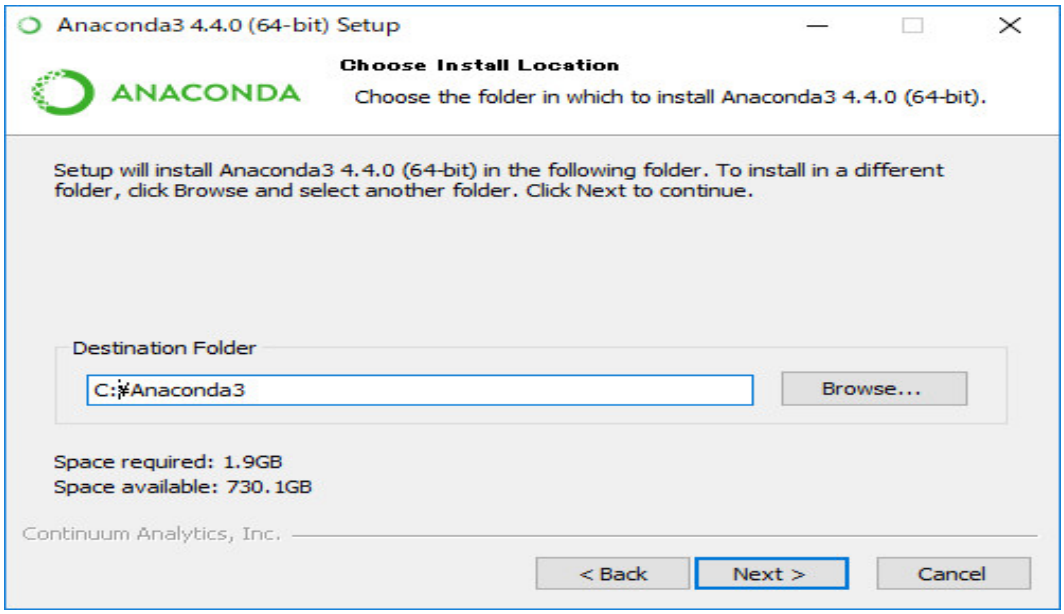
「Next」をクリックします。



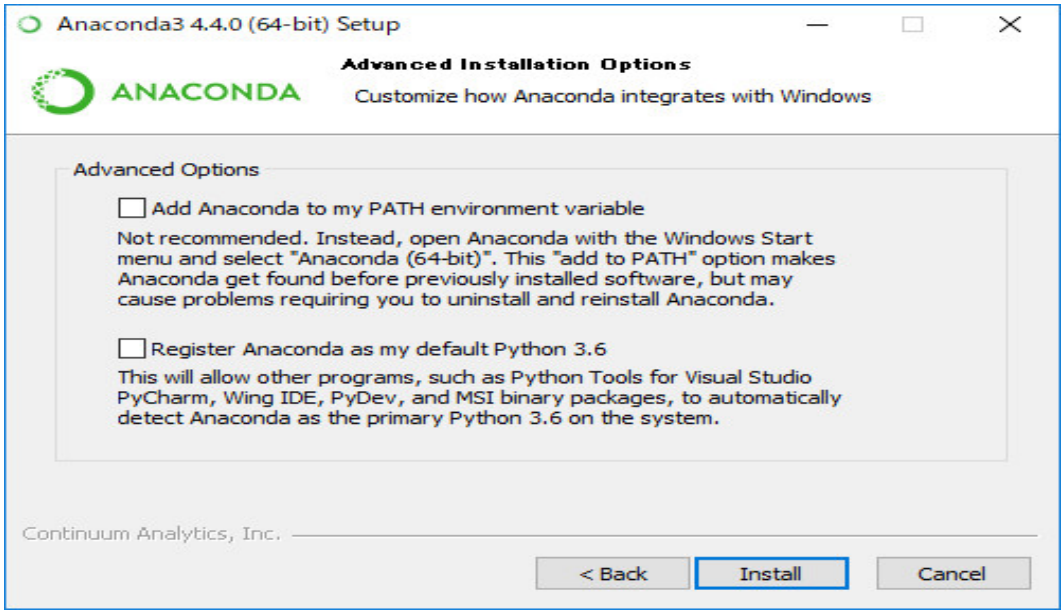
Destination Folder がデフォルトでは、後で色々問題が起こるのでここを

`C:\Anaconda3`

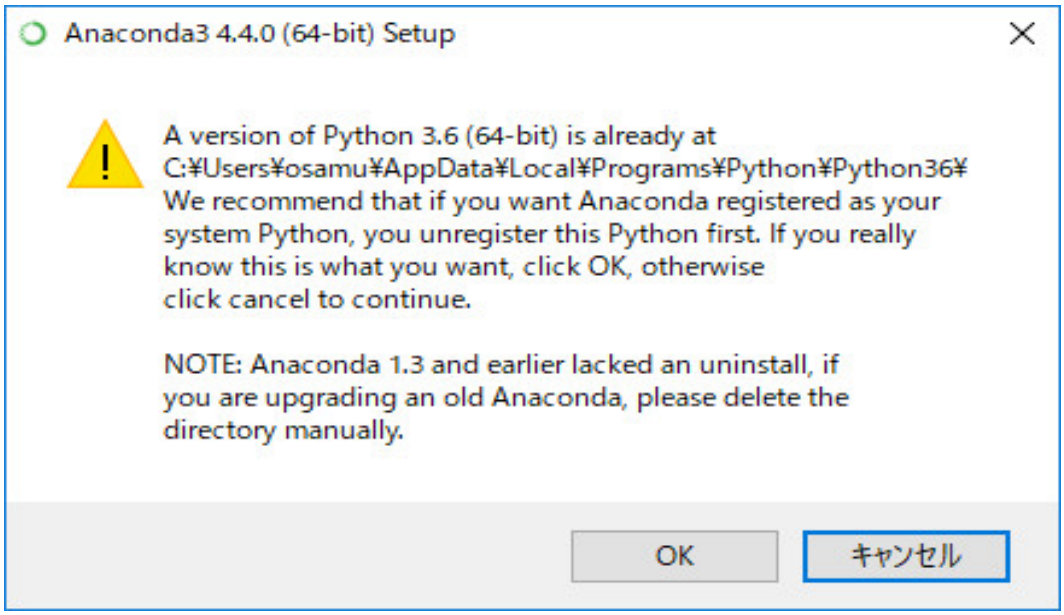
と修正します。



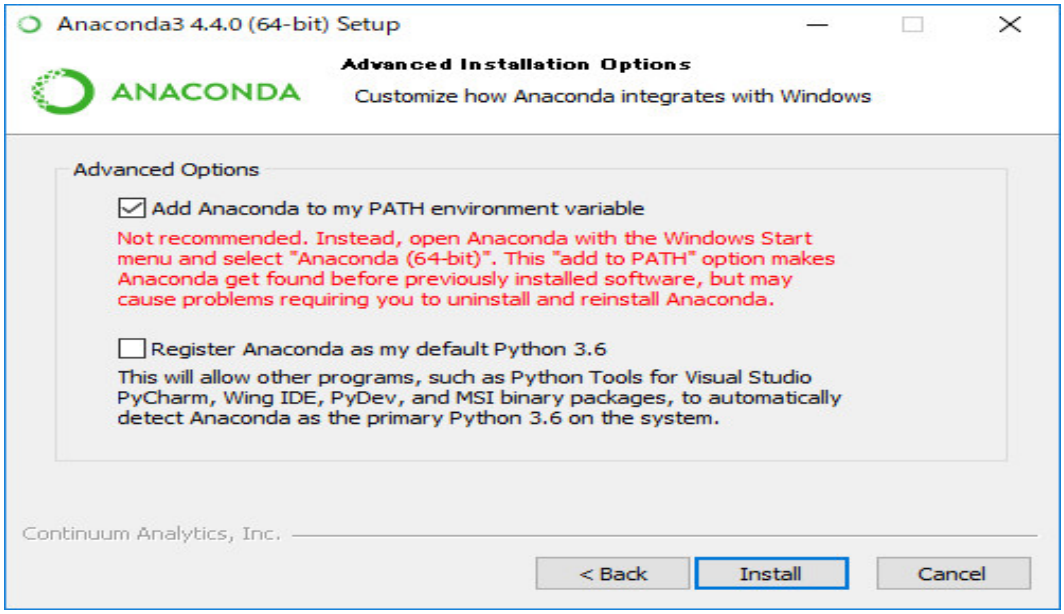
「Next」をクリックします。



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

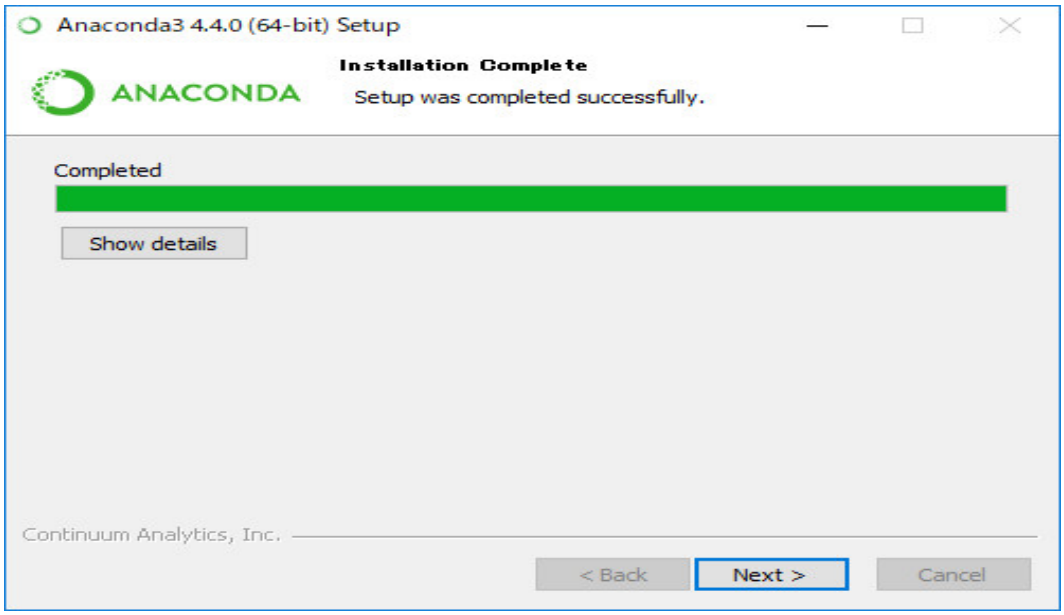


と表示されるので、「キャンセル」をクリックし、すでに Python3.6 をインストールしている場合には

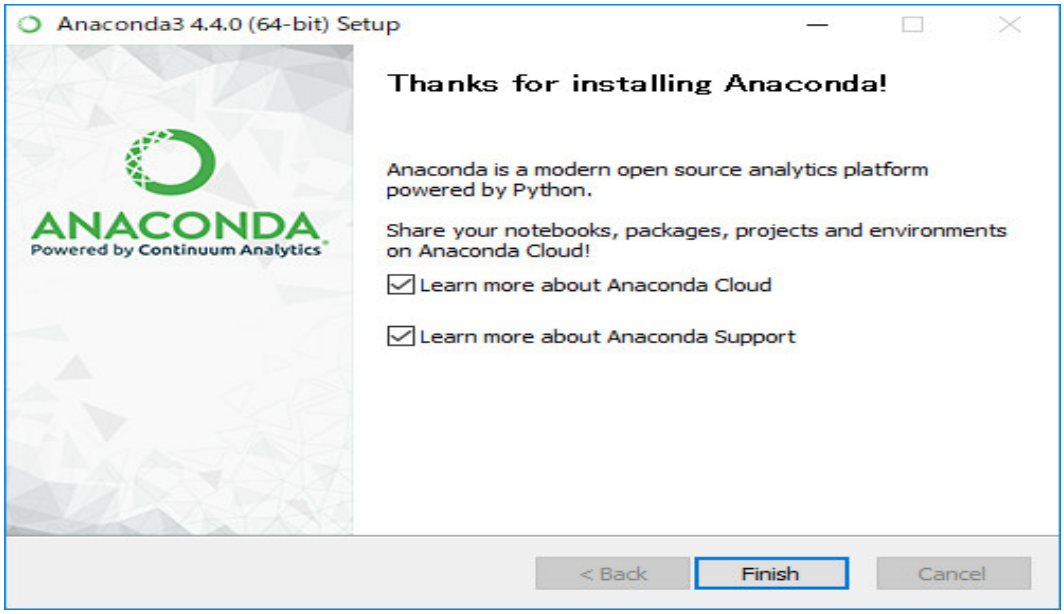


のように、上だけをチェックします。

「Install」をクリックします。

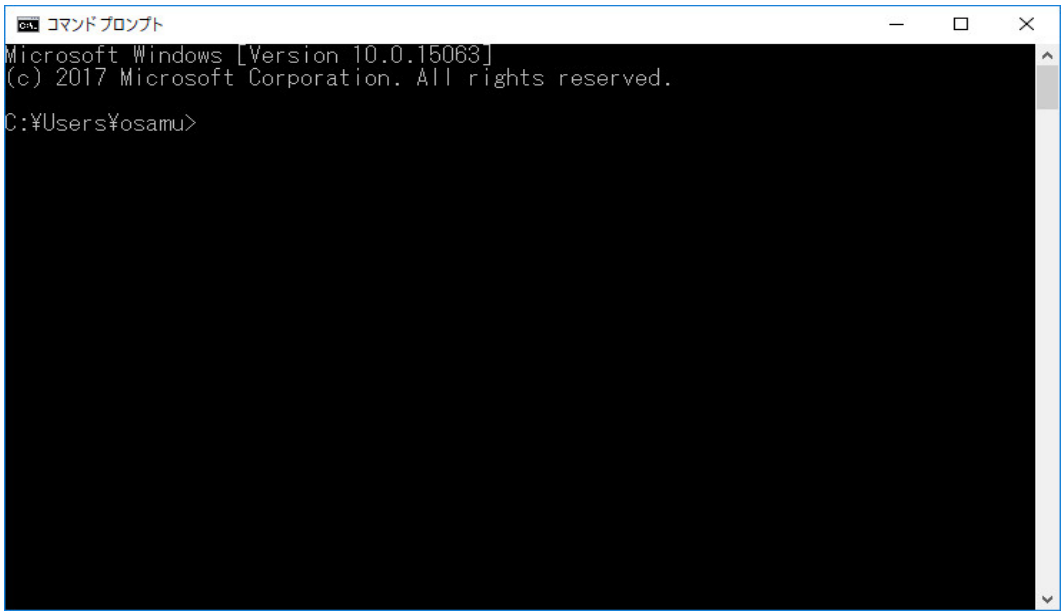


「Next」をクリックします。

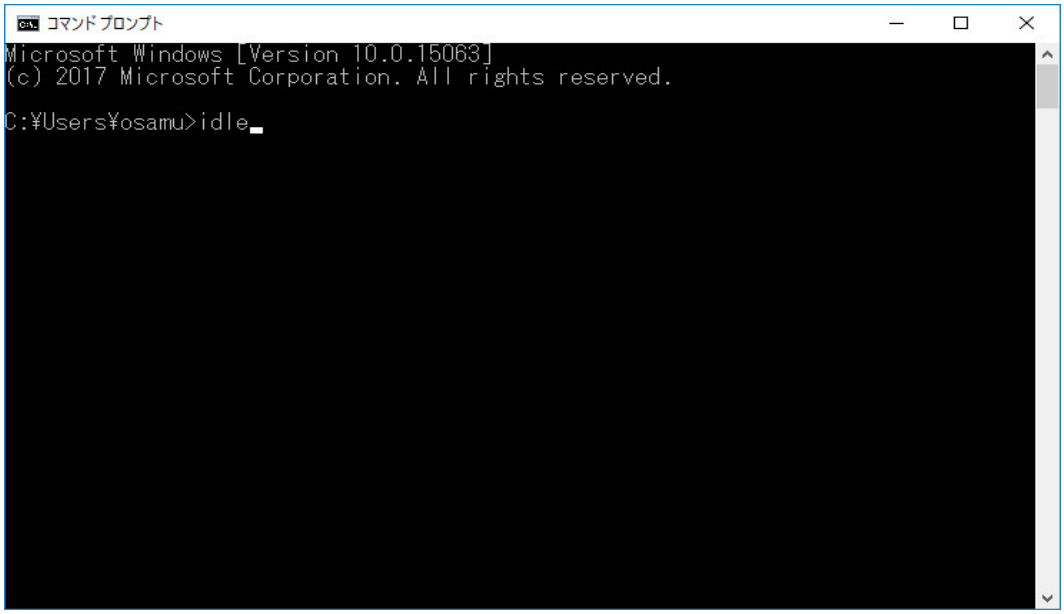


「Finish」をクリックします。

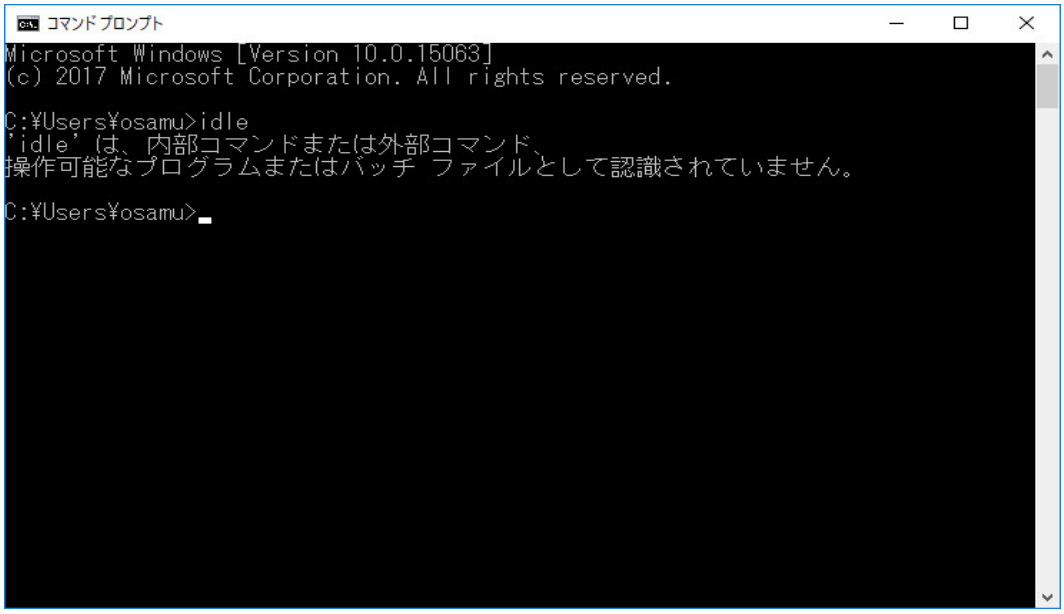
Anaconda をインストールした後に「コマンド　プロンプト」を起動し、



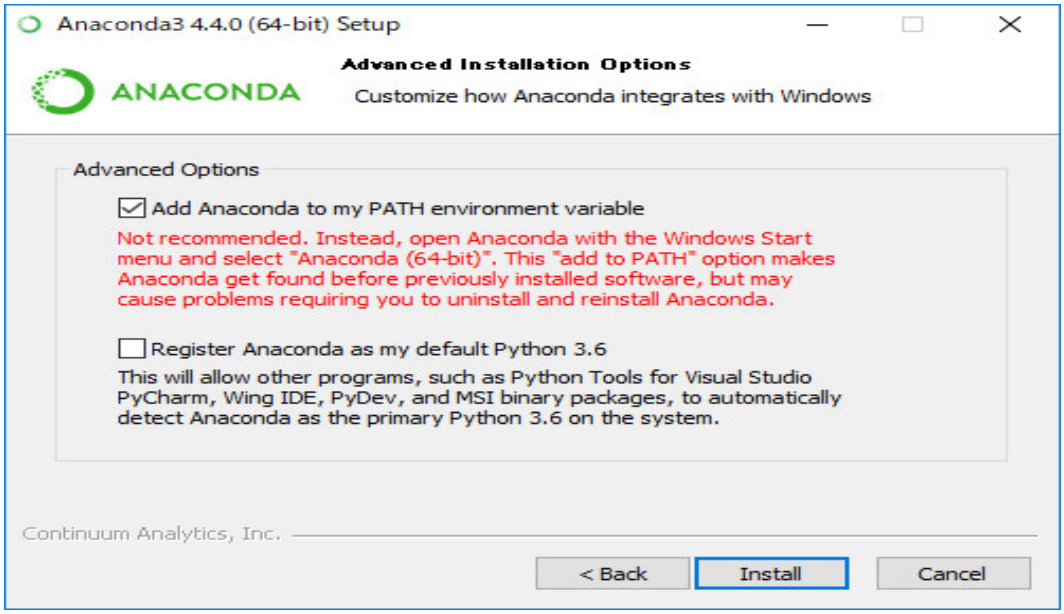
「idle」を



実行します。

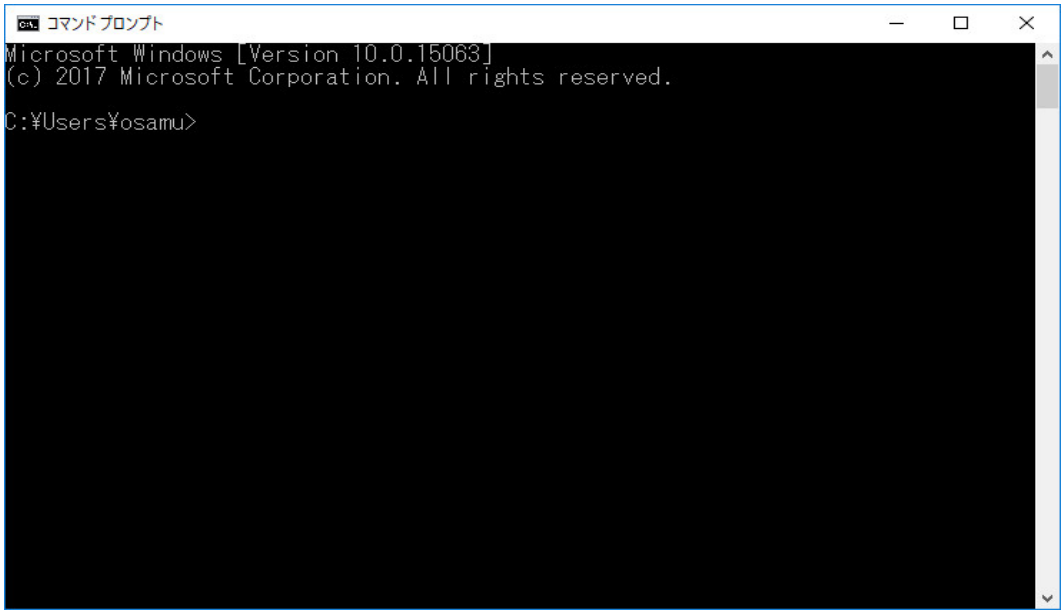


と表示されたら、

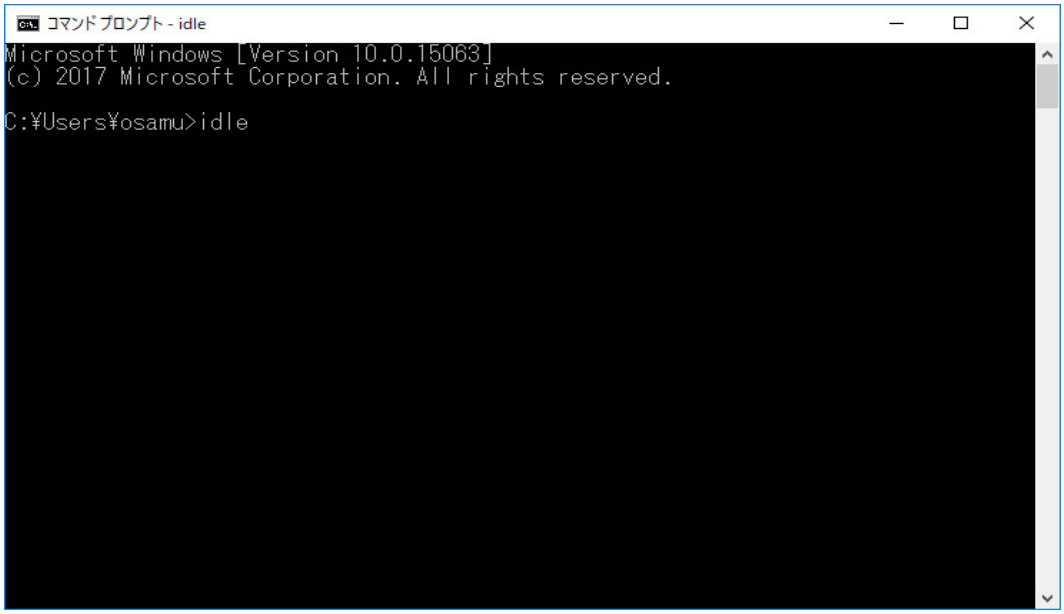


のチェックを忘れていた可能性があります。 Anaconda3 のフォルダーをエクスプローラーで削除して、もう一度インストールをやり直すのが簡単です。

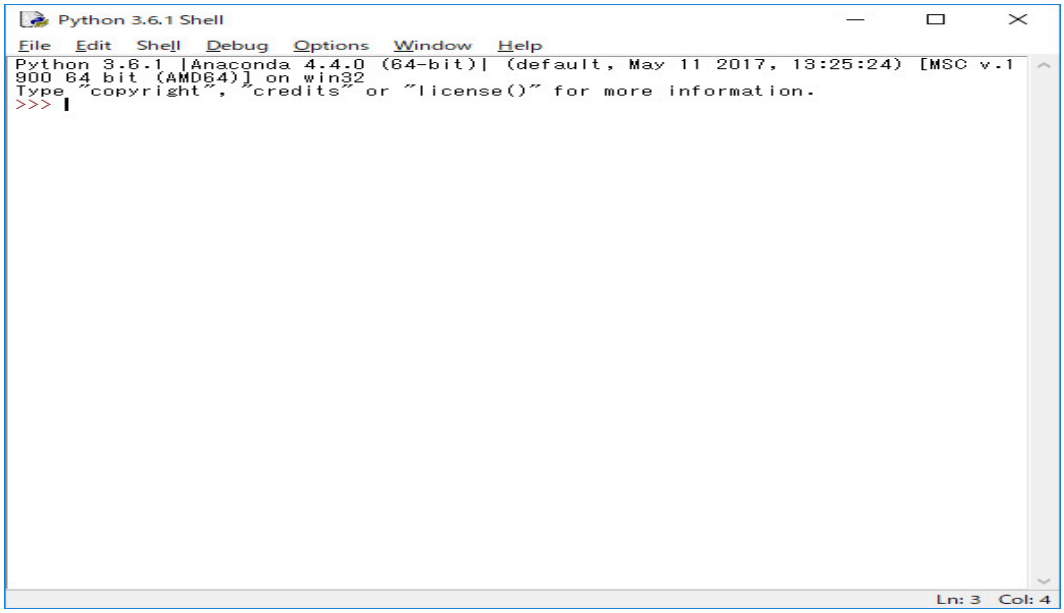
「コマンド　プロンプト」を起動し、



「idle」を

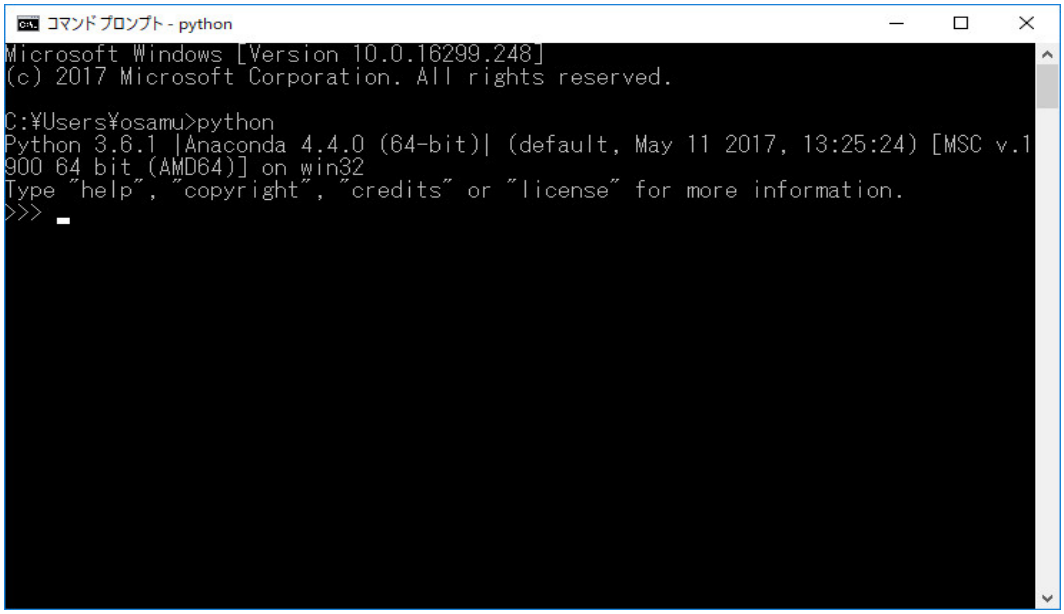


実行します。



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

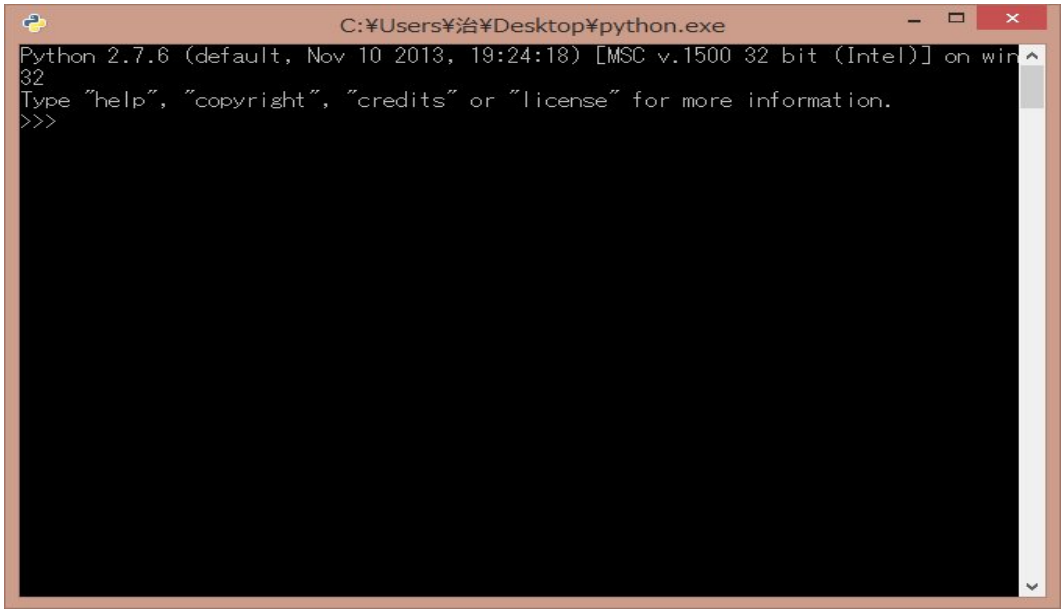
「コマンド　プロンプト」を起動し、「python」を



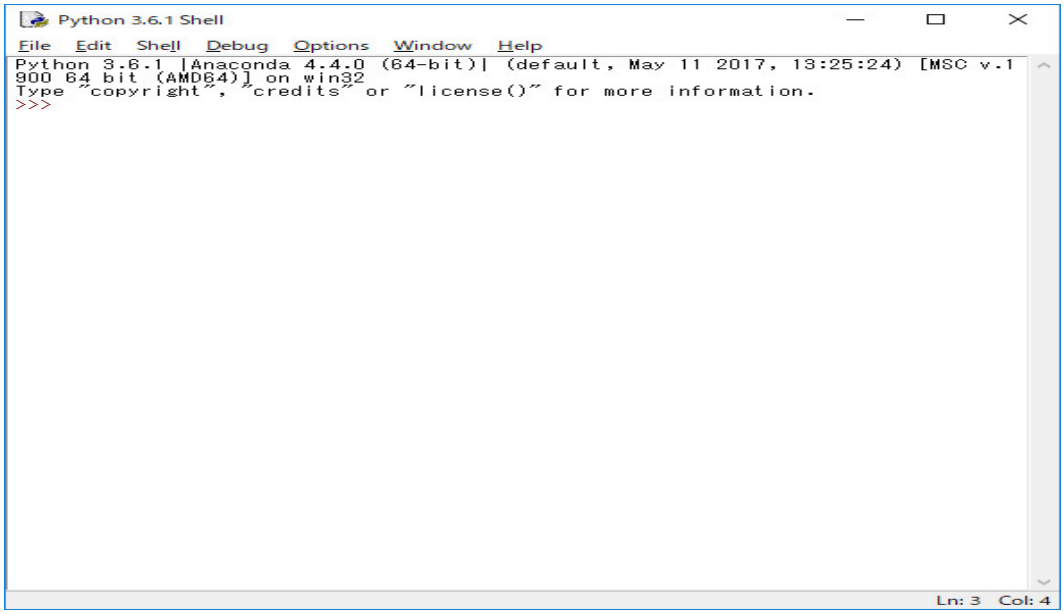
実行すると Python インタープリターが走り始めます。

では、Python によるタートルグラフィックスの講義を始めます。

コマンドプロンプトを起動し、idle.exe を実行します。



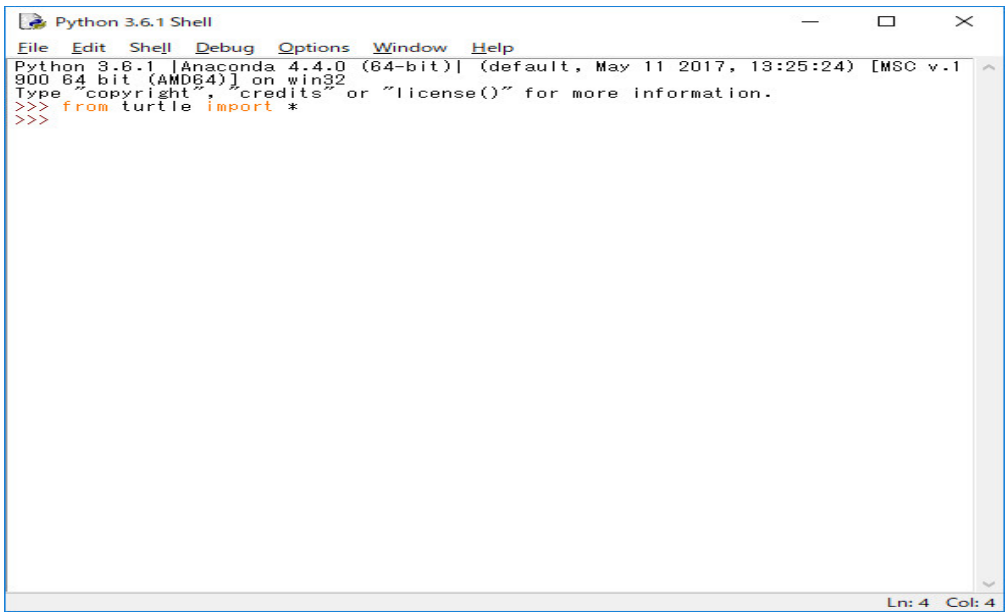
Python Shell のウィンドが開きます。



この Python の Shell に Python の命令を打ち込んでいきます。
>>> が Python のプロンプト（入力促進記号）です。
Turtle を使って絵を描いてみましょう。

```
from turtle import *
```

と打ち込みます。



何も起こりません。 Python のプロンプト（入力促進記号）>>> が再び表示されます。これで Python が標準装備しているタートルグラフィックス (turtle graphics) のモジュールが読み込まれ、タートルグラフィックスのすべての命令が実行できるようになりました。

```
from turtle import *
```

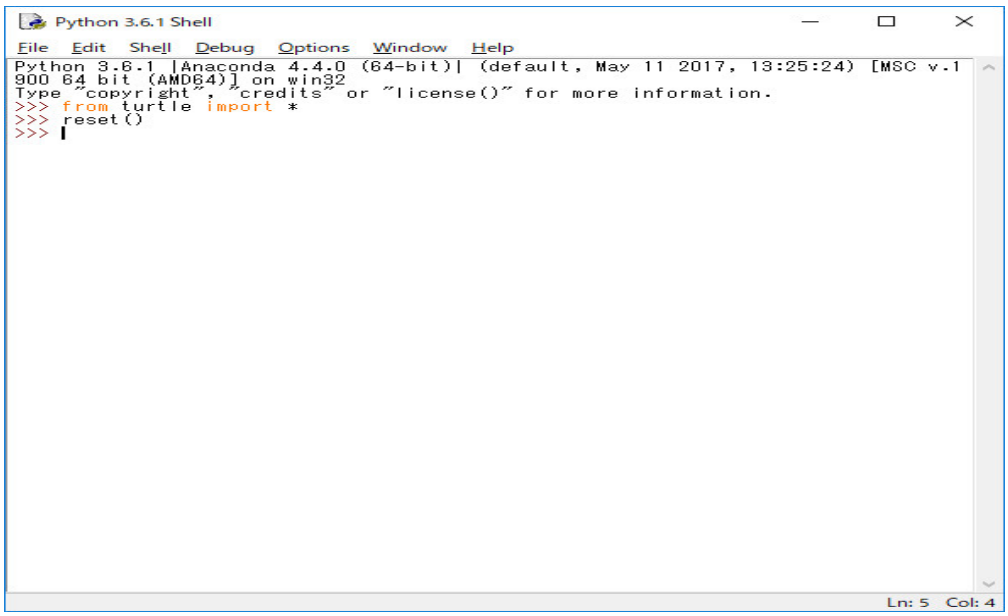
ではなく

```
import turtle
```

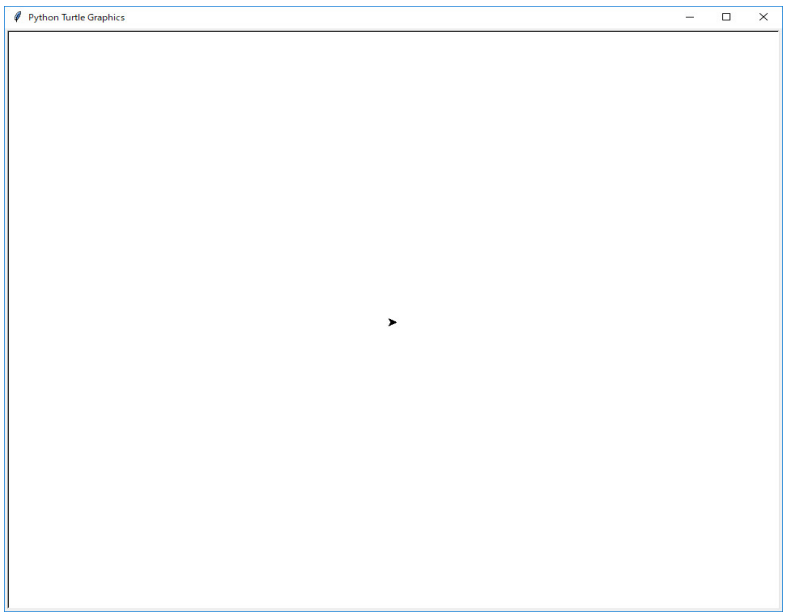
という命令でもタートルグラフィックスのすべての命令が実行できるようになりますが、命令の仕方が変化します。この違いは後程、説明します。

```
reset()
```

と打ち込みます。



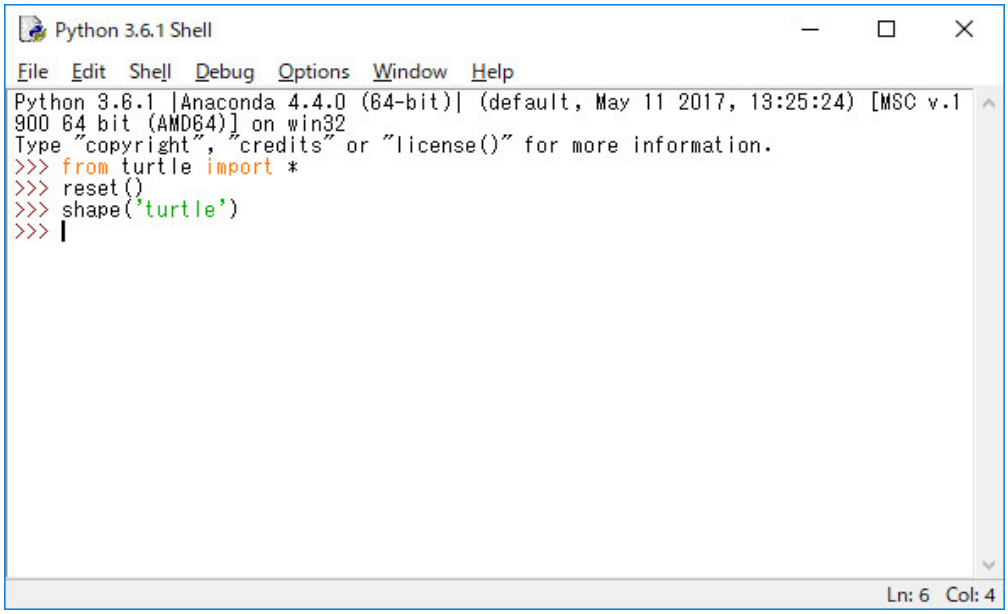
中央に矢印が表示された次のような新しいウィンドが表示されました。これが初期状態です。色々図を描いた後、reset() の命令を実行すると、線が消え、タートルが画面の中央に戻った、この画面になります。



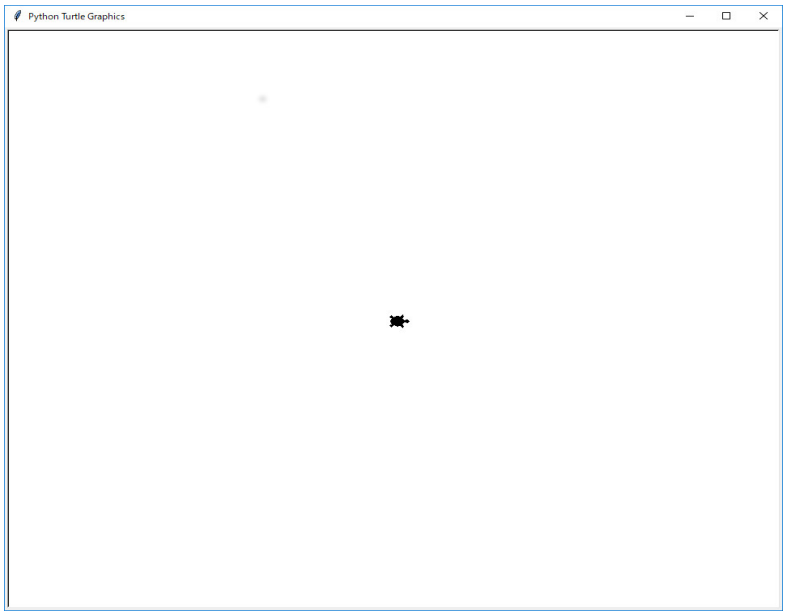
矢印を亀の絵に変えてみます。

```
shape('turtle')
```

と打ち込みます。



矢印の先頭の三角形がカメの絵に代わりました。このカメに命令して絵を描くのがタートルグラフィックス (turtle graphics) です。カメの絵を描くのに時間が掛かるのでデフォルトでは矢印が使われています。

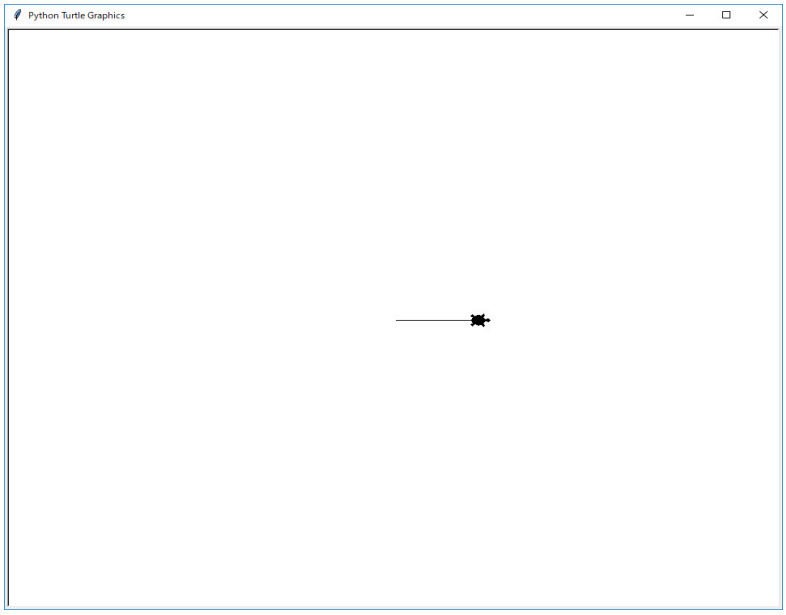


このタートル（亀）に命令して、動かしてみましょう。

```
forward(100)
```

と打ち込みます。

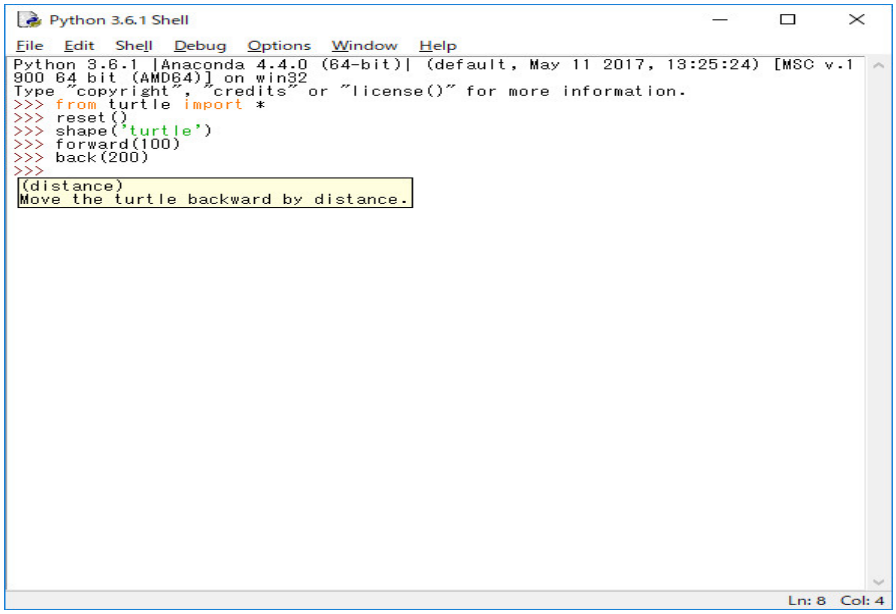
亀が右に１００ピクセル前進しました。

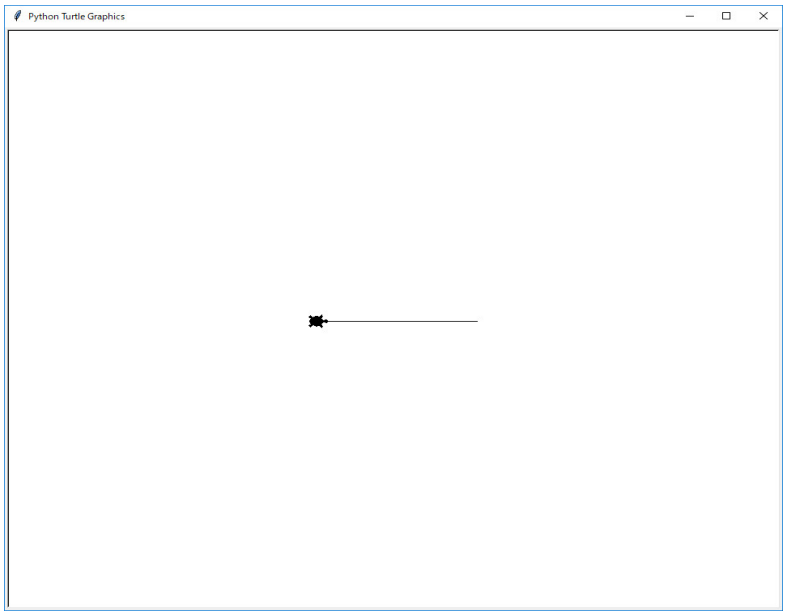


ここで使った forward(distance) という命令は、タートルが頭を向けている方へ、タートルを距離 distance だけ前進させます。亀の向きは変えません。fd(distance) という省略形も使えます。

```
back(200)
```

と打ち込みます。



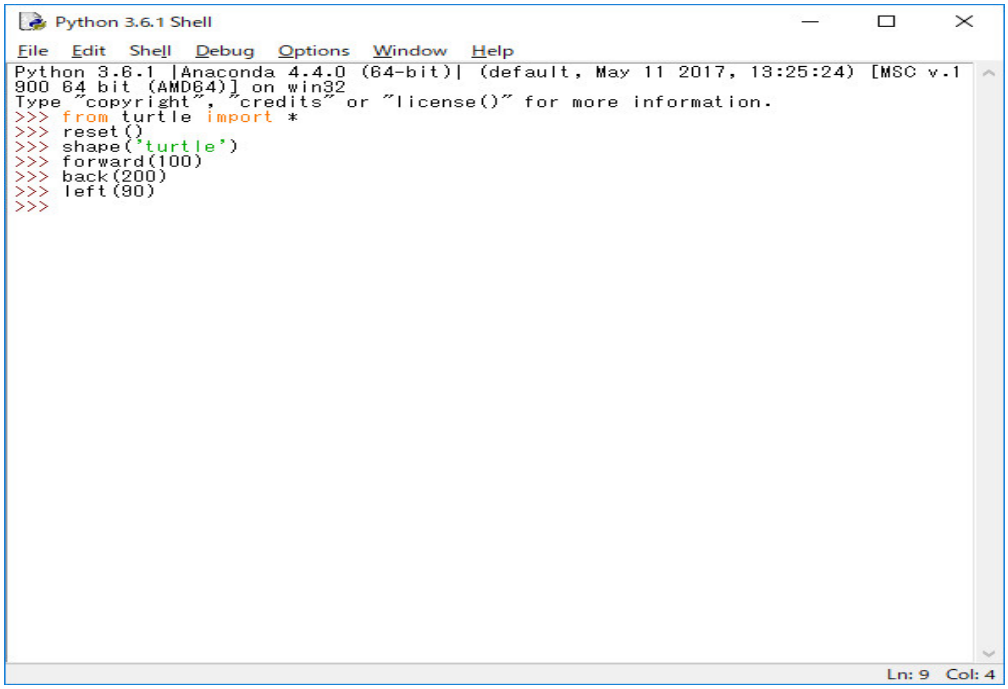


ここで使った back(distance) という命令はタートルが頭を向けている方と反対方向へ、タートルを距離 distance だけ前進させます。タートルの向きは変えません。bk(distance) という省略形も使えます。この back(distance) という命令は、 forward(distance) という命令に負の distance を与えることでも実現できます。同様に、 back(distance) という命令に負の distance を与えることで、 forward(distance) という命令の代用ができます。

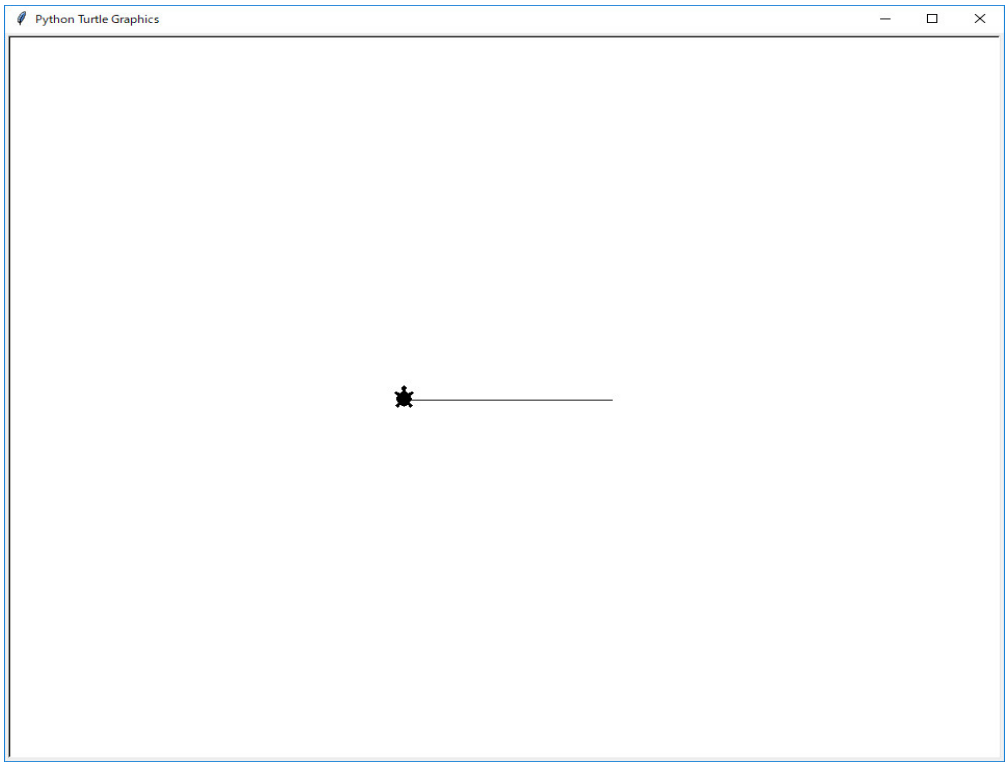
亀の向きを変えるには right(angle) か left(angle) という命令を使います。 right(angle) は亀を angle 単位だけ右に回します。 left(angle) は亀を angle 単位だけ左に回します。単位のデフォルトは度です。rt(angle) と lt(angle) という省略形も使えます。同じく angle に負の数値を代入することで左右の回転の向きが変わります。

```
left(90)
```

と打ち込みます。



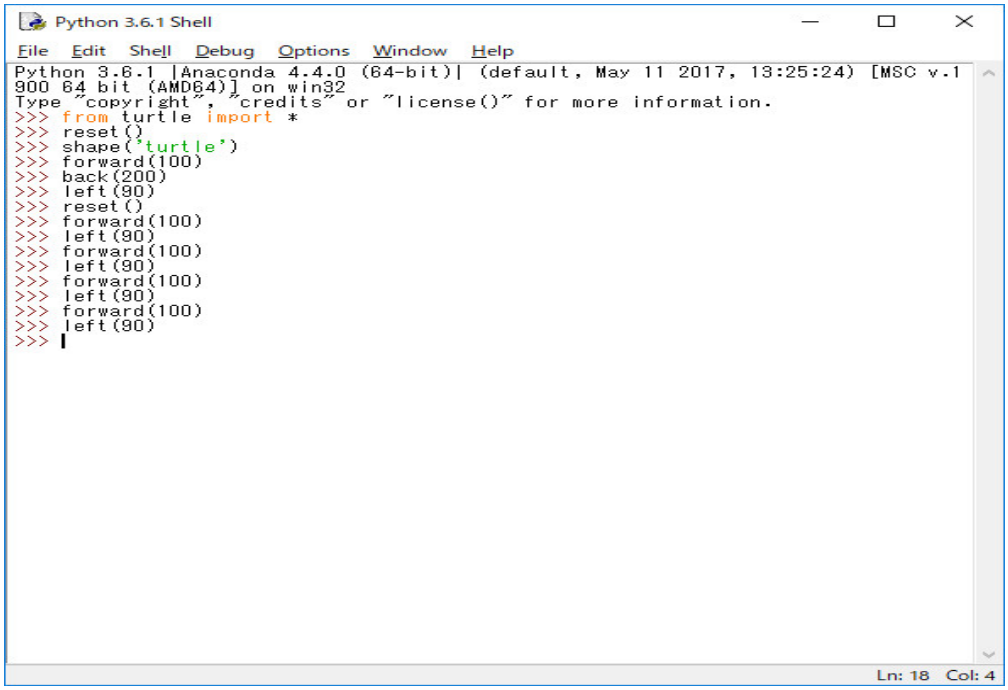
亀が左に 90 度回転し、上を向きました。



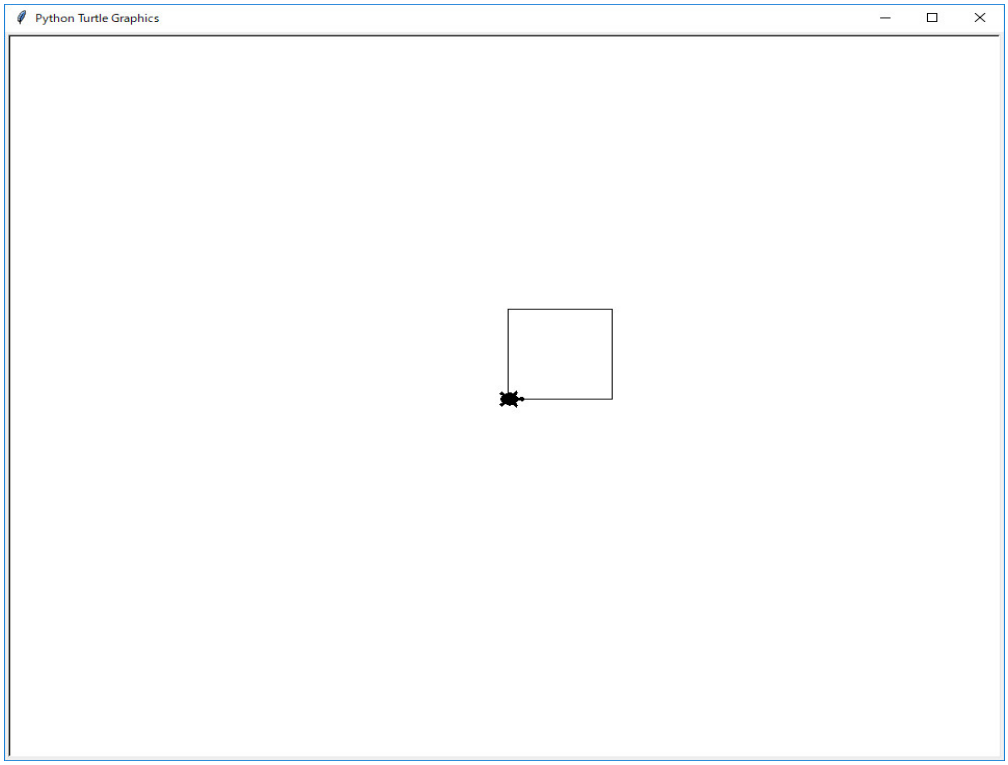

```
reset()
forward(100)
```

```
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
```

と打ち込みます。



次の図のように正方形を描いて、亀が元の状態に戻りました。



同じ命令を何度も繰り返しています。このような場合は繰り返しの命令を使います。まず for 文の使い方を学びます。

```
for i in range(4):
```

と打ち込みます。最後にコロン　：　を打つのを忘れないようにしてください。Enter キーを押すと勝手に字下げしてくれます。まだ命令が完結してないので、続けて

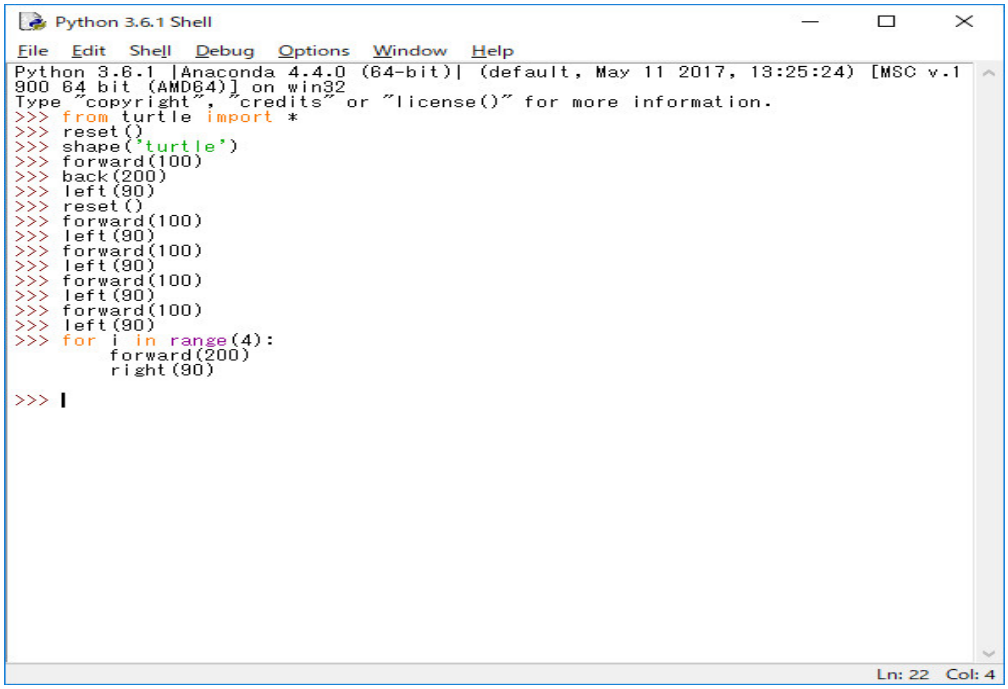
```
    forward(200)
```

と打ち込みます。まだ、命令は完結していません。Enter キーを押すと勝手に字下げしてくれます。

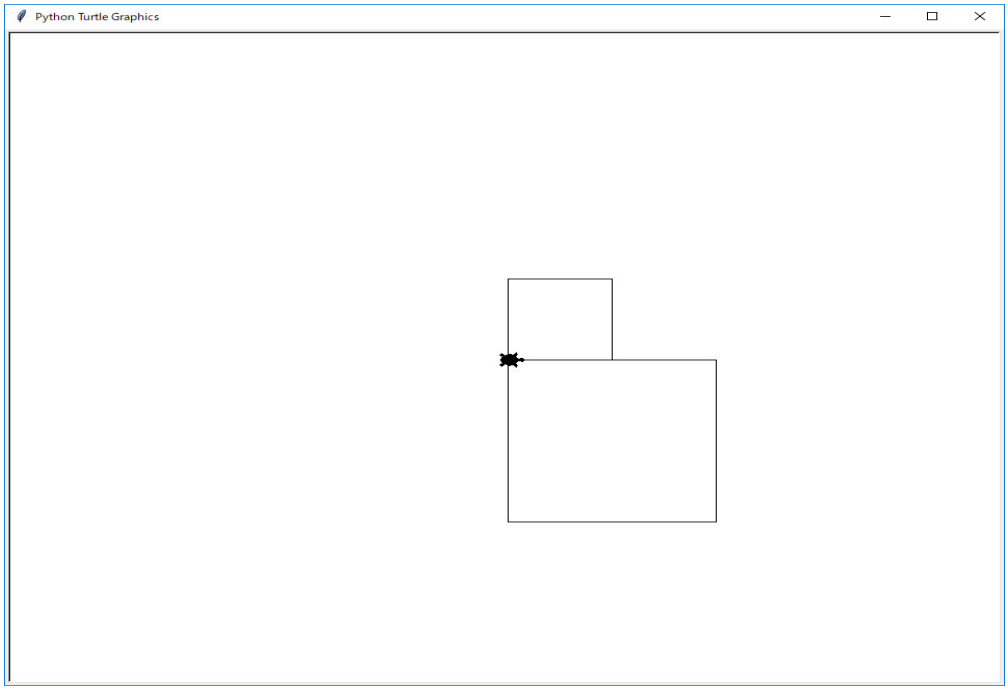
```
    right(90)
```

と打ち込みます。

これで命令が完結したので、 Back Space キーを押し、for 文が終わったことをコンピュータに指示して、Enter キーを押します。



一辺　200 の正方形を書きました。




```
for i in range(4):
    forward(200)
    right(90)
```

は for 文と言われるものです。i が 0, 1, 2, 3 と変化しながら、字下げされた範囲の命令 forward(200) と right(90) を計 4 回繰り返します。

Python 2.7 では

```
>>> range(4)
```

と打ち込むと

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

と表示されていましたが、Python 3.6 以降では単に range(0, 4) と表示されるだけです。

```
>>> list(range(4))
```

と list() 函数を使ってリストに変換してやると

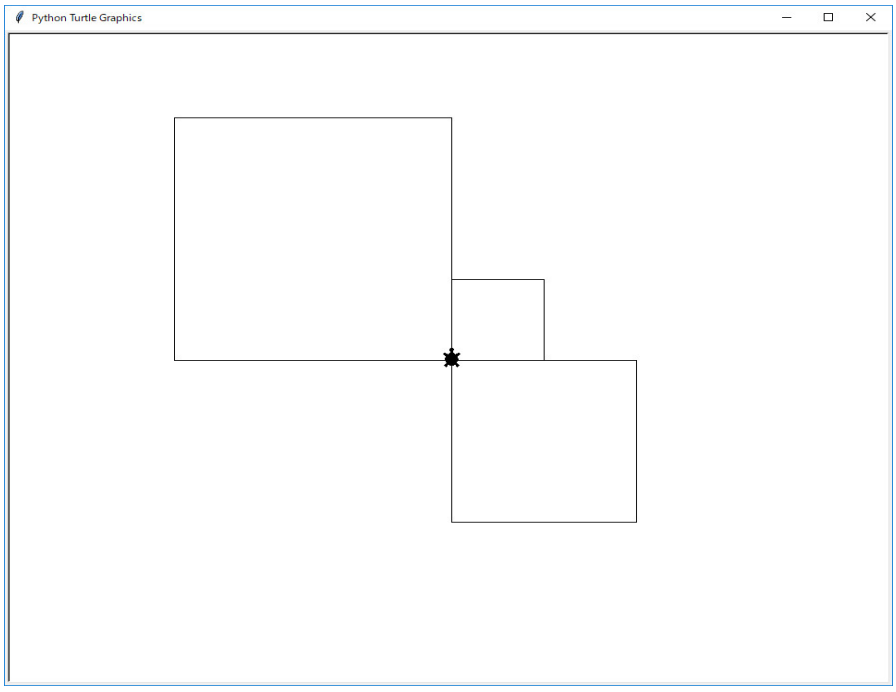
```
>>> list(range(4))
[0, 1, 2, 3]
```

と表示します。

[0, 1, 2, 3] のようなものはリストと呼ばれます。0 と 1 と 2 と 3 を要素とする長さ 4 のリストです。for i in range(4): はこのリスト [0, 1, 2, 3] に含まれる要素をひとつづつ i に代入し、インデント（字下げ）された部分の命令を各 i に対して一回ずつ実行しなさいという命令です。range(4) の代わりに、[0,1,2,3] を使っても同じ結果が得られます。range(n) は $[0, 1, \cdots, n-1]$ と同じです。ここで n は正整数です。

```
left(90)
for i in [0,1,2,3]:
    fd(300)
    lt(90)
```

を実行してみましょう。以下、実行結果のみ示します。



```
for i in range(3):
    print(i)
```

と打ち込むと

```
0
1
2
```

と表示されます。ここで i は変数で、値を入れて置く箱のようなもので、一般に英文字＋英数字の列で良いです。

range() 関数の書式は次の通りです。

```
range([start,] stop[, step])
```

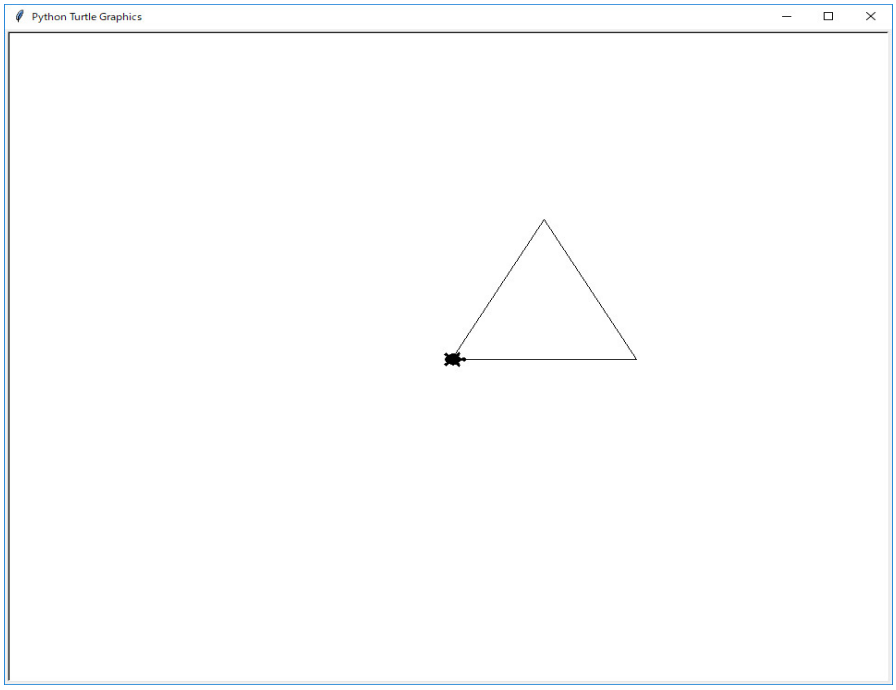
引数は全て整数で指定します。「stop」だけを指定した場合には、0 から「stop - 1」まで順に 1 ずつ増加する要素を持つリストの対応するものが作成されます。開始する数値を指定する場合は引数「start」を指定して下さい。range(2, 5) なら [2, 3, 4] と同じです。デフォルトでは次の要素は前の要素に 1 を加えた数値となりますが、引数「step」を指定すると指定した値だけ次の要素には追加されます。この時最後の値は「stop」より小さい最大の数値となります。range(1, 6, 2) なら [1, 3, 5] と同じです。各引数の値は負の整数も指定可能です。引数「step」が負の値の場合、最後の値は「stop」より大きい最小の数値となります。range(0, -5, -2) なら [0, -2, -4] と同じです。

さて、タートルグラフィックスに戻って、画面を初期化し、正三角形を描いてください。

解答は

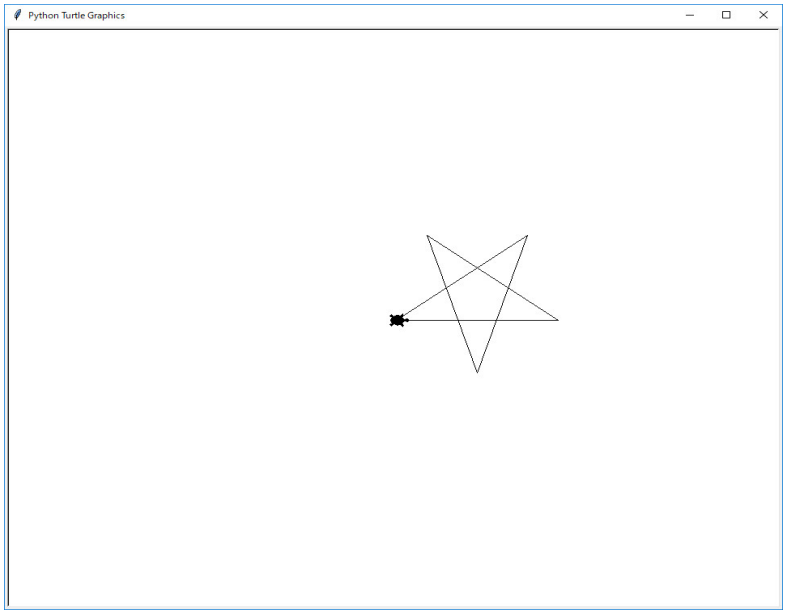
```
reset()
for i in range(3):
    forward(200)
    left(120 )
```

です。left(60) ではないので注意してください。この場合、内角ではなく、外角が問題になります。




```
reset()
for i in range(5):
    forward(200)
    left(360*2/5)
```

を実行すると星形正五角形を描きます。



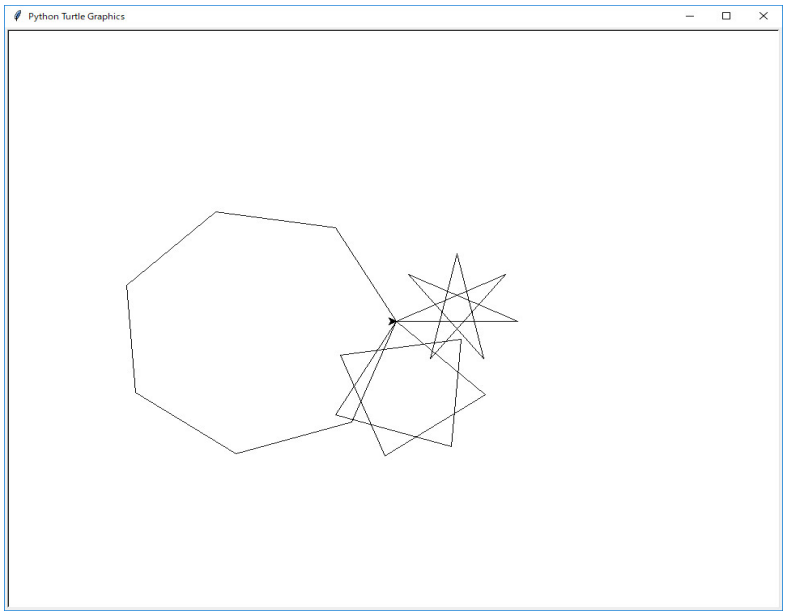
left(360*2/5)　の * と / は掛け算と割り算です。足し算は + で、引き算は- です。
問題：

三種類の正 7 角形（普通の正 7 角形と二種類の星形正 7 角形）を描くにはどうすれば良いか述べなさい。

解答例は

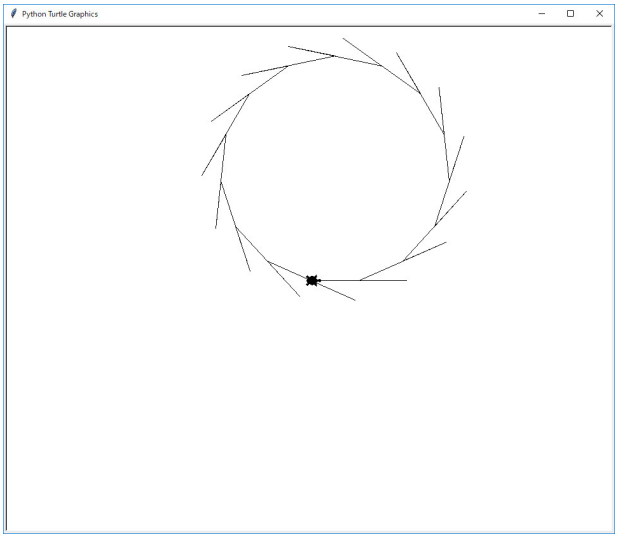
```
reset()
for i in [1,2,3]:
    lt(120)
    for k in range(7):
        fd(150); lt(360*i/7)
```

です。for 文はこのようにネスト（for 文の中に for 文を書くこと）することが出来ます。更に、セミコロン ; を間に置くことで複数の命令を一行に書くことが出来ます。$360 * i/7$ $(i = 1, 2, 3)$ がそれぞれの正 7 角形の外角を与えています。



問題：

次のような図を描くプログラムを作れ。



解答例は

```
reset()
for i in range(15): fd(150); fd(-75); lt(24)
```

です。

円や円弧を描くには

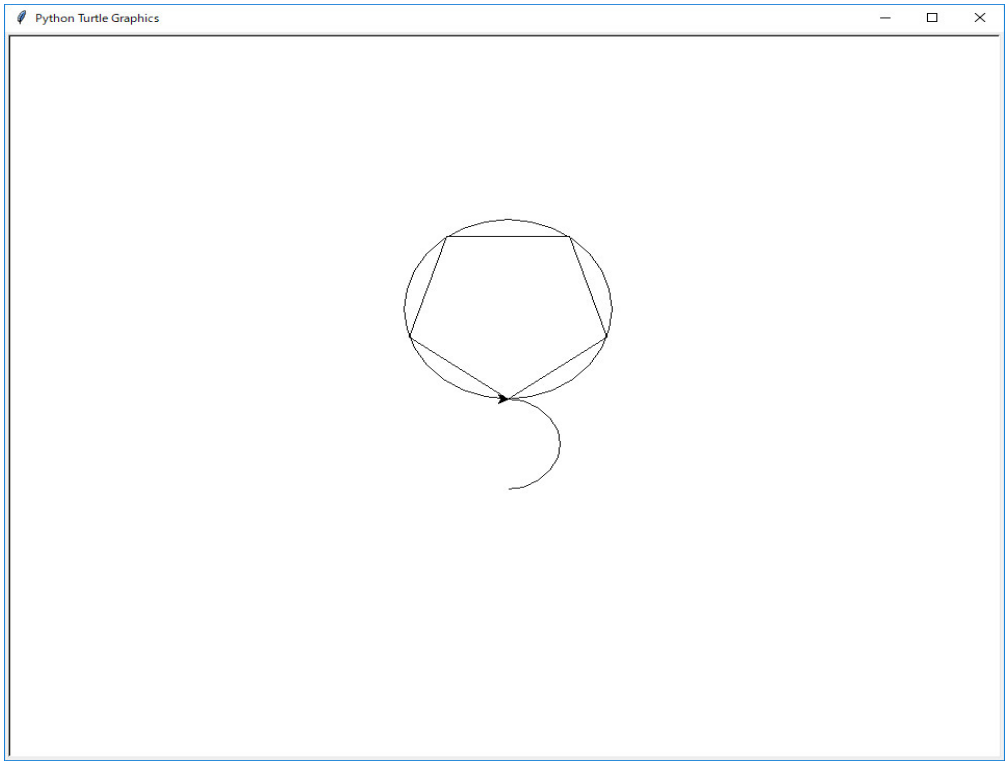
```
circle(radius, extent=None, steps=None)
```

という circle 命令が準備されています。 circle(radius) は半径 radius の円を描きます。中心はタートルの左 radius ユニットの点です。extent （角度です） は円のどの部分を描くかを決定します。extent が与えられなければ、デフォルトで完全な円になります。extent が完全な円でない場合は、弧の一つの端点は、現在のペンの位置です。radius が正の場合、弧は反時計回りに描かれます。そうでなければ、時計回りです。最後にタートルの向きが extent 分だけ変わります。

円は内接する正多角形で近似されます。steps でそのために使うステップ数を決定します。この値は与えられなければ自動的に計算されます。また、これを正多角形の描画に利用することもできます。

```
reset()
circle(100)
circle(-50, 180)
penup()
home()
pendown()
circle(100, 360, 5)
```

を実行する。



ここで出てきた新しい命令は penup() , pendown(), home() です。それぞれ次のような意味です。

penup() は pu(), up() と省略でき、ペンを上げます 。 タートルが動いても線は引かれません。

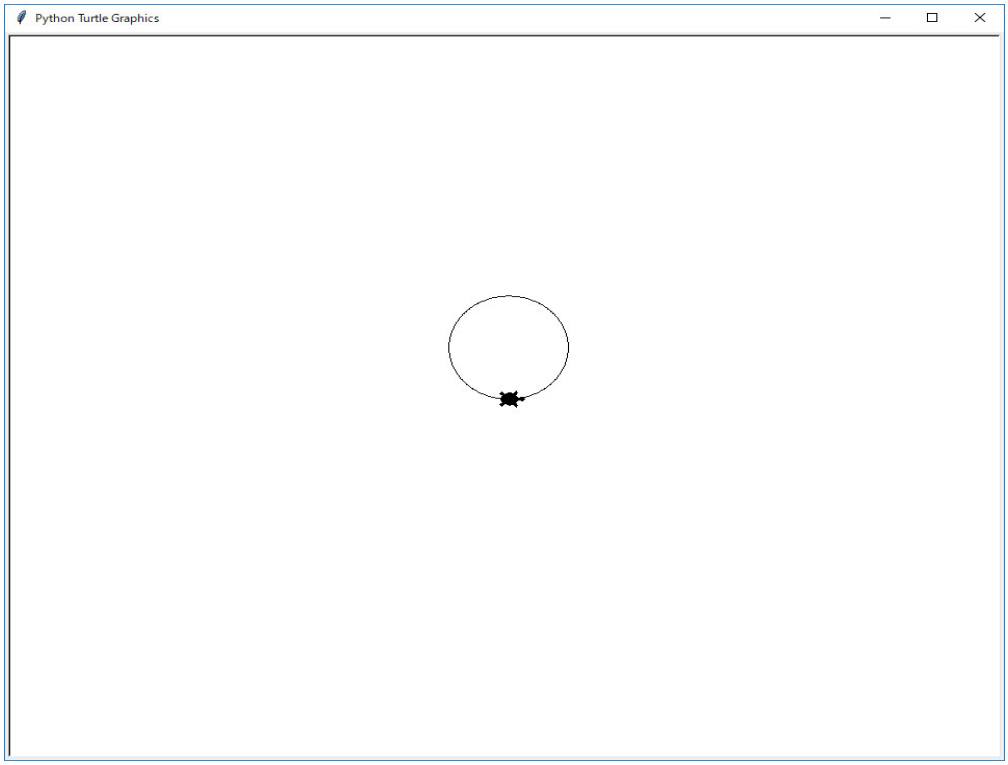
pendown() は pd(), down() と省略でき、ペンを下ろします。 動くと線が引かれます。初期状態ではペンが下りています。

home() はタートルを原点 （座標 (0, 0) ：画面の中心）に移動し、向きを開始方向に設定します。

実は円を描くのに circle() 命令を使わなくてもいいです。正多角形（例えば、正 360 角形）で近似すれば良いです。

```
for i in range(360):
    fd(1)
    lt(1)
```

とすれば、時間はかかりますが、円を描いてくれます。

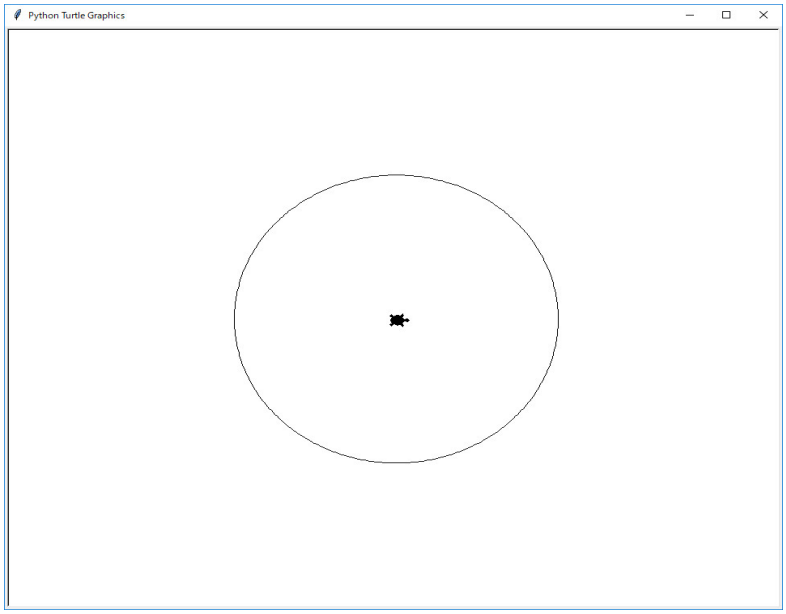


LOGO はこのようにして円を描いていました。現在のタートルの位置を中心とする半径 R=200 の円を描くには、円周の長さは $2\pi R$ ですから、正 360 角形の一辺は $2\pi R/360$ で近似できます。π の値は、変数を使って $PI = 3.14159265$ としても良いですが、Python には準備されています。math をインポート (import math か from math import * とする）すれば、最初の場合 math.pi か後者の場合単に pi で円周率が利用できます。math をインポートすれば、各種の数学函数（三角関数、逆三角関数、指数関数、対数関数など）が利用できます。必要になれば、紹介します。Python では関数名や変数名は大文字と小文字を区別します。PI, Pi, pi はそれぞれ別のものとされます。

従って、

```
import math
R = 200
pu(); fd(R); pd(); lt(90)
for i in range(360): fd(2*math.pi*R/360); lt(1)
pu(); rt(90); bk(R); pd()
```

で良いです。



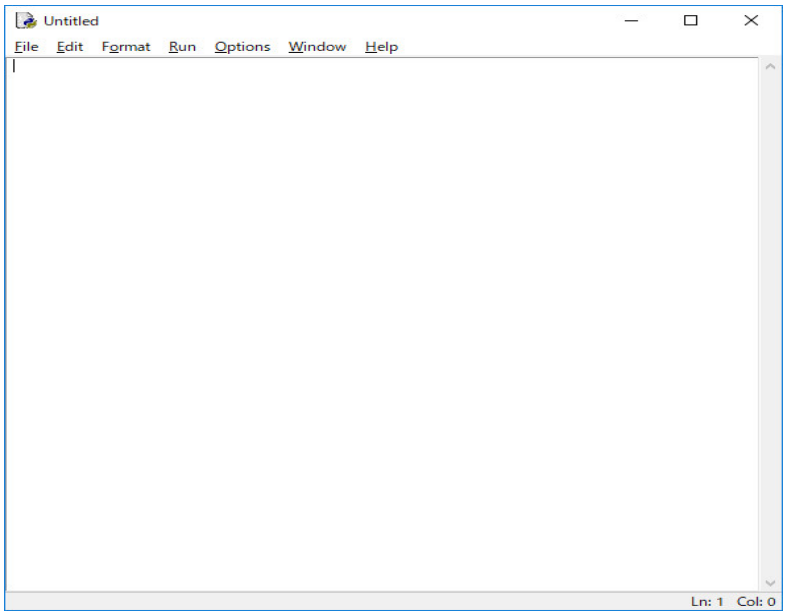
となります。

問題：

半円や扇型を描くプログラムを作れ。

一般的な半径 r 、中心角 angle の扇型を描くプログラムは色々考えられますが、プログラムが複雑になってきたので、今までのように一行命令ずつ実行し、結果を見ながら実行するのではなく、全体のプログラムを作ってから、実行するようにします。こうすれば間違えたときの処理が簡単になり、またパラメータを色々変えて実行するのにも都合がよくなります。

Python の Shell の File メニューの New File をクリックします。

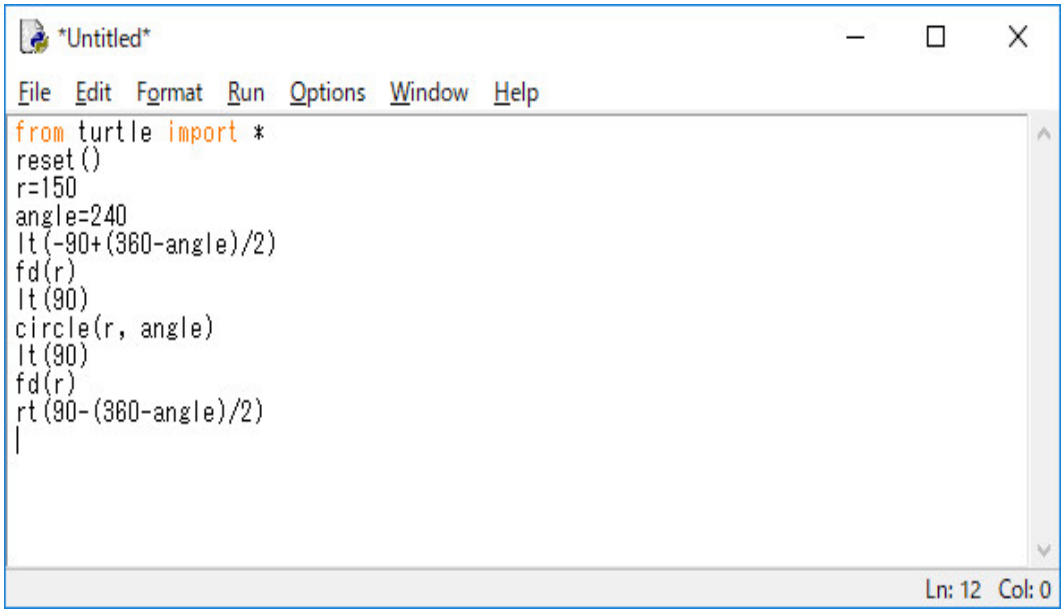


Python のプログラムを打ち込むためのエディタが開きます。ここにプログラムを打ち込みます。

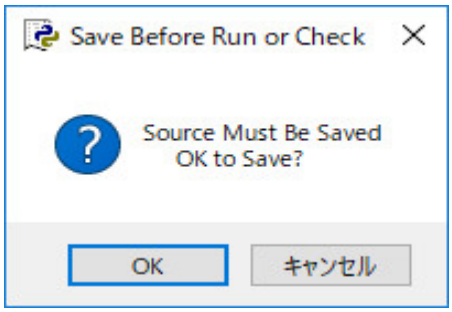
ここでは問題の解答例として

```
from turtle import *
reset()
r=150
angle=240
lt(-90+(360-angle)/2)
fd(r)
lt(90)
circle(r, angle)
lt(90)
fd(r)
rt(90-(360-angle)/2)
```

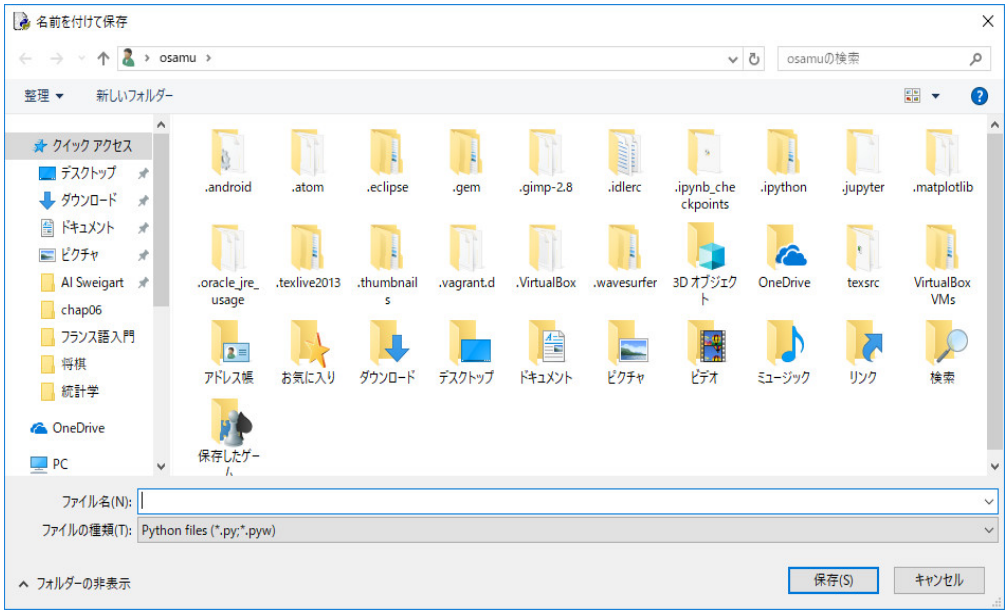
と打ち込みます。



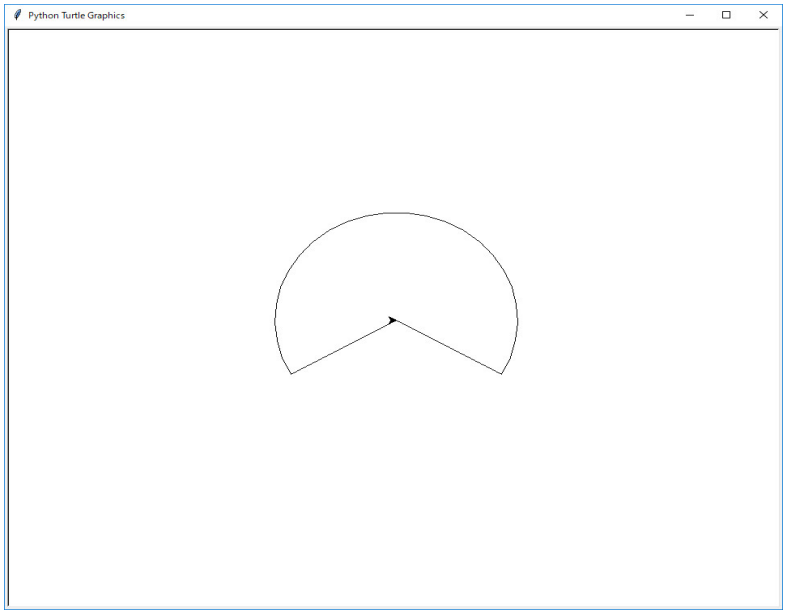
できたら、Run メニューの Run Module をクリックします。



OK のボタンをクリックします。



適当なフォルダをえらび、適当な名前を付けて、このプログラムを保存します。プログラムが間違っていなければ



となります。ここで、例えば、最後の命令が

```
tt(90-(360-angle)/2)
```

となっていれば、Shell に

```
Traceback (most recent call last):
  File "C:/PythonSRC/oogi.py", line 11, in <module>
    tt(90-(360-angle)/2)
NameError: name 'tt' is not defined
```

のようなエラーの表示がなされるので、それを読んでプログラムを修正します。この例では、11 行目にある tt() という命令は定義されていないと言っています。

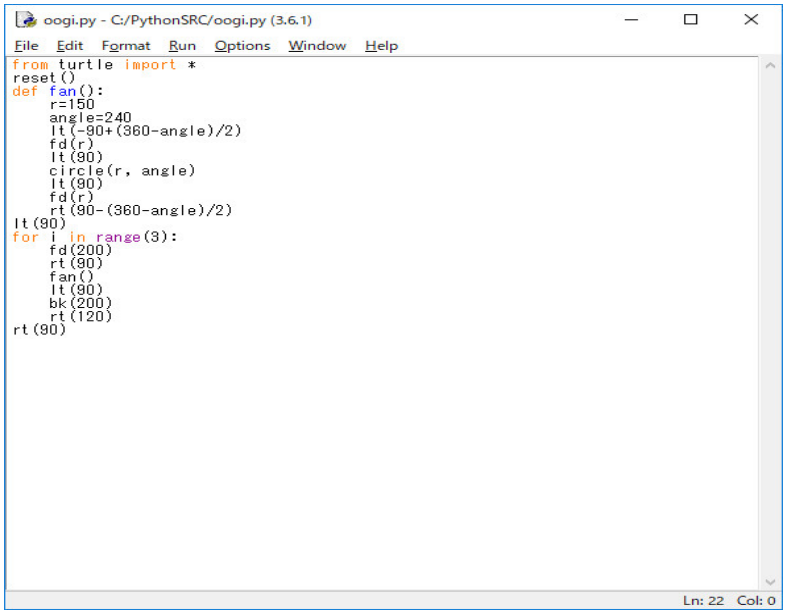
このプログラムでは、円の接線は半径と 90 度で交わることを使っています。中心角とタートルの進むべき方向の関係を図示して考えれば、このプログラムがしていることが理解できると思います。そして、扇形を描くだけなら、タートルを書き始めの状態に戻しておくことは余計な手間だと考えるかもしれませんが、こうしておくとせっかく作ったこのプログラムを有効活用できます。

それでは、このプログラムを使って、もう少し複雑な絵を描いてみましょう。そのためには、このプログラムに名前を付け、fd() 命令のように、いつでも呼び出せるようにします。

まず単純に

```
from turtle import *
reset()
def fan():
    r=150
    angle=240
    lt(-90+(360-angle)/2)
    fd(r)
    lt(90)
    circle(r, angle)
    lt(90)
    fd(r)
    rt(90-(360-angle)/2)
lt(90)
for i in range(3):
    fd(200)
    rt(90)
    fan()
    lt(90)
    bk(200)
    rt(120)
rt(90)
```

とプログラムを修正します。




```
def fan():
    r=150
    angle=240
    lt(-90+(360-angle)/2)
    fd(r)
    lt(90)
    circle(r, angle)
    lt(90)
    fd(r)
    rt(90-(360-angle)/2)
```

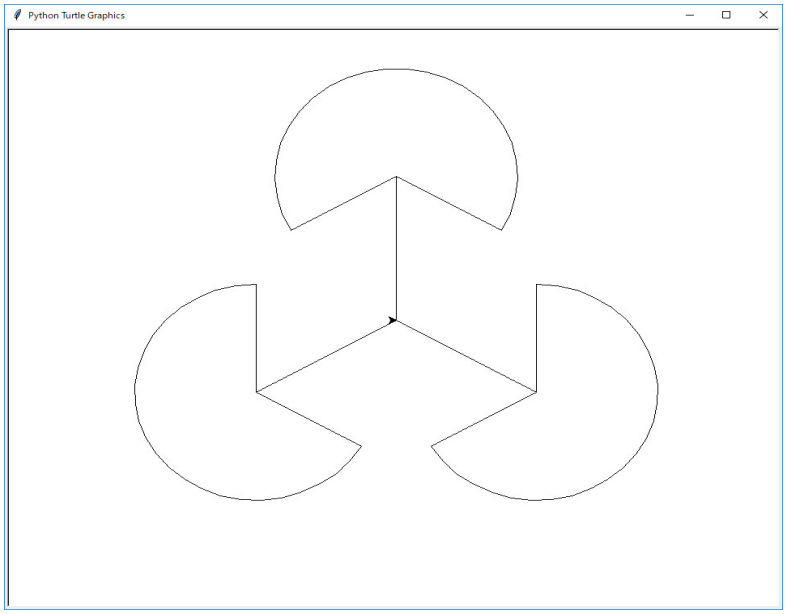
で、上で作った扇形をプログラムを def fan(): の下にそのまま打ち込んでいます。

```
def 関数名 ():
```

で、自分の関数を作ることが出来ます。今の場合、fan() とすれば、半径 150、中心角 240 度の扇形を描いてくれます。後半の

```
lt(90)
for i in range(3):
    fd(200)
    rt(90)
    fan()
    lt(90)
    bk(200)
    rt(120)
rt(90)
```

はこの自前の関数 fan() を組み込んだプログラムです。実行すると



のような絵を描きます。この fan() では, 扇形の大きさが決まっています。これを変えることが出来るように修正します。そのためには、fan() の中で決めている半径 r の値と中心角 angle の値を fan() を

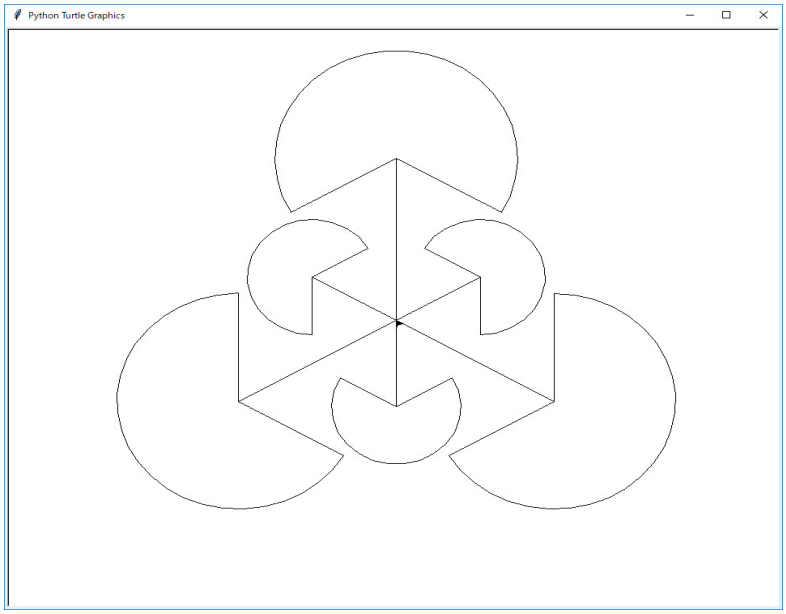
```
def fan(r, angle):
    lt(-90+(360-angle)/2)
    fd(r)
    lt(90)
    circle(r, angle)
    lt(90)
    fd(r)
    rt(90-(360-angle)/2)
```

と、引数 r と angle を持つ関数として定義し、r と angle の値を関数の外で与えてやるようにします。ここで、数学では関数の変数といいますが、プログラミングでは引数という言い方をします。fan(150, 240) と打ち込めば、元の fan() と同じ働きをします。この新しい fan() 函数を使って、もっと複雑な絵を描いてみましょう。

```
from turtle import *
reset()
def fan(r, angle):
    lt(-90+(360-angle)/2)
    fd(r)
    lt(90)
    circle(r, angle)
    lt(90)
    fd(r)
    rt(90-(360-angle)/2)
```

```
lt(90)
for k in [150, 80]:
    for i in range(3):
        fd(k*1.5)
        rt(90)
        fan(k, 240)
        lt(90)
        bk(k*1.5)
        rt(120)
    lt(60)
rt(90)
```

とプログラムを修正すると、

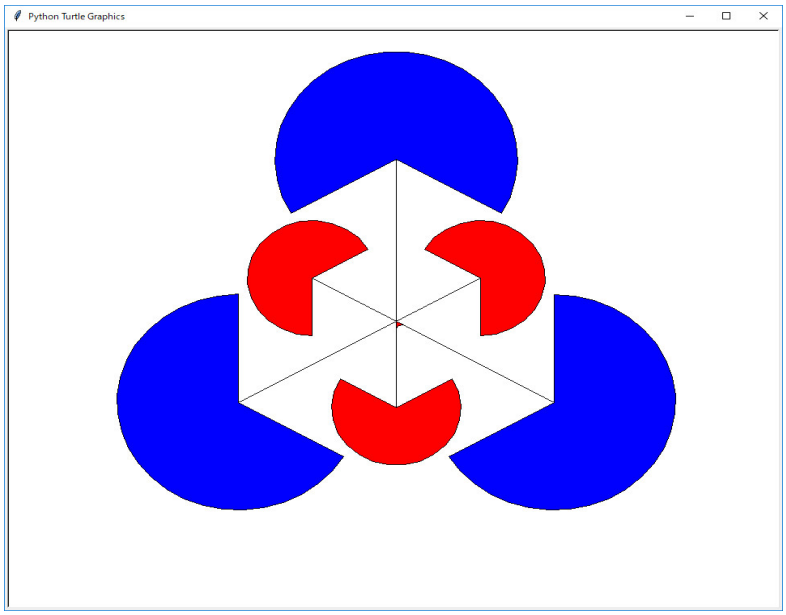


を描きます。扇形を色を付けて塗ってみたくなりました。

```
from turtle import *
reset()
def fan(r, angle):
    lt(-90+(360-angle)/2)
    fd(r)
    lt(90)
    circle(r, angle)
    lt(90)
    fd(r)
    rt(90-(360-angle)/2)
lt(90)
for k, c in [(150, 'blue'), (80, 'red')]:
    for i in range(3):
```

```
        fd(k*1.5)
        rt(90)
        fillcolor(c)
        begin_fill()
        fan(k, 240)
        end_fill()
        lt(90)
        bk(k*1.5)
        rt(120)
    lt(60)
rt(90)
```

と修正し、実行すると



を描きます。

まず、

```
for k, c in [(150, 'blue'), (80, 'red')]:
```

ですが、 [(150, 'blue'), (80, 'red')] ですが、これは 2 つのタプル (150, 'blue') と (80, 'red') を要素とするリストです。タプルはリストは [と] で要素を囲っていましたが、タプルは (と) で要素を囲います。リストとタプルの大きな違いはタプルは一度作ると変更できないことです。使い方はおいおい見ていきます。この for 文でタプル (150, 'blue') と (80, 'red') が順次取り出され、タプルの最初の値が k に、次の値が c にセットされ、for 文の支配範囲の命令が実行されます。'blue' や 'red' は文字列で、この文字列そのものを表します。

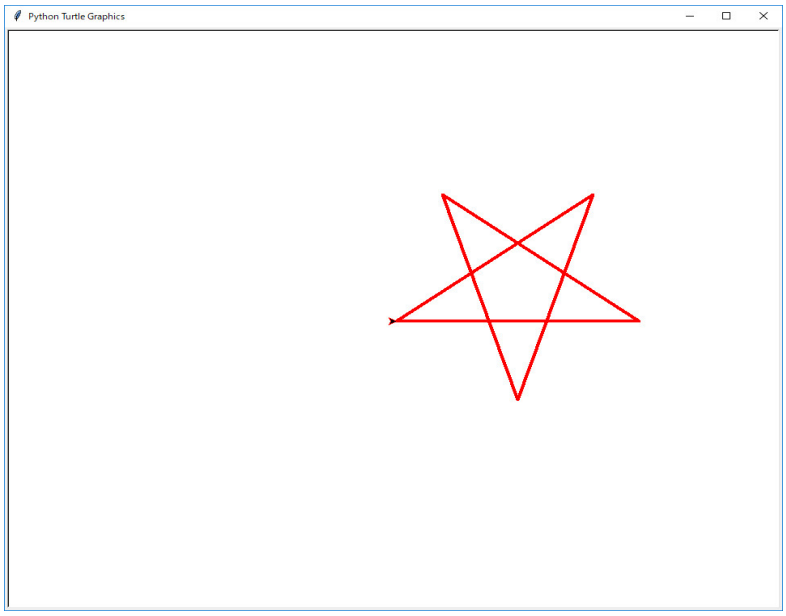
```
        fillcolor(c)
        begin_fill()
        fan(k, 240)
        end_fill()
```

の部分で、塗りつぶしの色と塗りつぶす範囲を指示しています。fillcolor(c) が塗りつぶす色の指定で、for 文の最初のステップで fillcolor('blue') 、つぎのステップで、fillcolor('red') と指定されます。そして `begin_fill()` と `end_fill()` で囲まれた部分の図形が指定した色で塗りつぶされます。

また

```
from turtle import *
reset()
pencolor("red")
pensize(4)
for i in range(5):
    fd(300); lt(360*2/5)
```

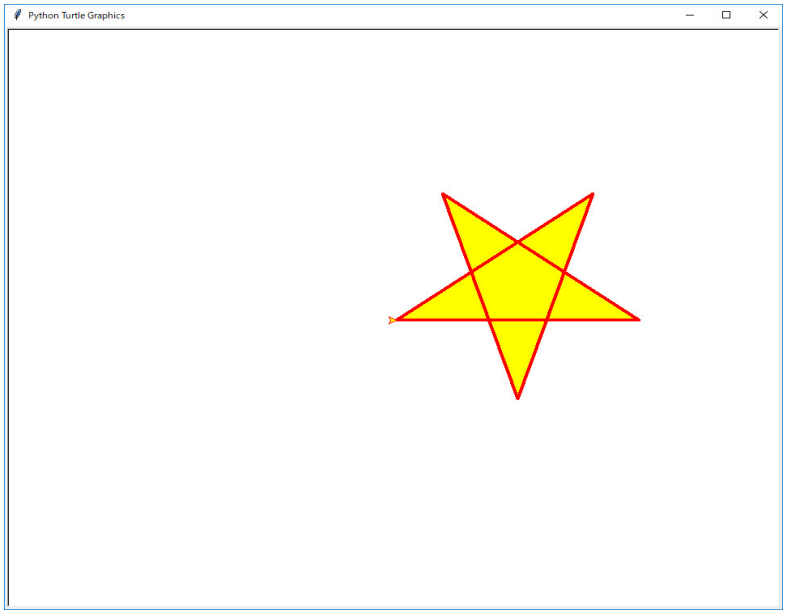
のプログラムで



を描きます。pencolor("red") でペンの色を指定しています。文字列は " で囲んでも良いです。pensize(4) で描く線のサイズを指定しています。

```
from turtle import *
reset()
color("red", ''yellow'')
pensize(4)
begin_fill()
for i in range(5):
    fd(300); lt(360*2/5)
end_fill()
```

のプログラムで



を描きます。color(”red”, ”yellow”) 命令で線の色と塗りつぶす色を一度に指定できます。
　円に内接する正多角形を簡単に描くには、例えば

```
circle(100)
circle(100, 360, 5)
```

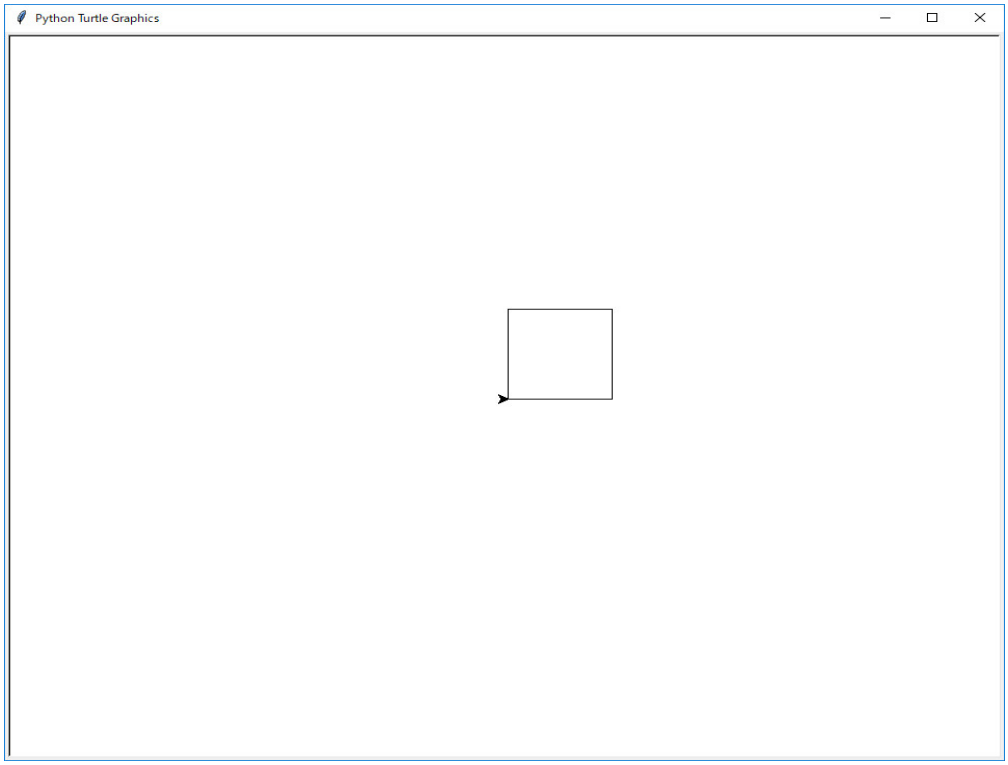
とすれば良いですが、

```
circle(100, 360, 5)
```

を使わずに、更に、三角関数の知識も使わずに、中学生でもできる方法で、円に内接する正多角形を描く方法を探求してみましょう。
　setpos() という命令を使えば、座標の知識があれば、図を描くことが出来ます。

```
setpos(100, 0)
setpos(100, 100)
setpos(0, 100)
setpos(0, 0)
```

とすれば、一辺 100 の正方形を描きます。




```
setpos(100, 0)
setpos(100, 100)
```

の代わりに

```
setpos([100, 0])
setpos((100, 100))
```

とすることも出来ます。

setpos() の引数はリスト (list) [x, y] でも タプル (tuple) (x,y) でも良いです。[と] で要素を囲んだものがリストですが、(と) で要素を囲んだものがタプルです。リストとタプルの違いは、リストは要素を書き換えることが出来ますがタプルは要素を書き換えることが出来ません。

座標を使って正方形を描くには、リストのリスト [[100 ,0], [100, 100], [0, 100], [0, 0]] を使って次のようにすることも可能です。

```
pos_list = [[100, 0], [100, 100], [0, 100], [0, 0]]
for x in pos_list:
    setpos(x)
```

これは `pos_list` すなわち、リスト [[100, 0], [100, 100], [0, 100], [0, 0]] から順番に x に [100, 0],[100, 100], [0, 100], [0, 0] を代入し、setpos(x) を実行します。つまり、

```
setpos([100, 0])
setpos([100, 100])
setpos([0, 100])
setpos([0, 0])
```

を実行します。
　リストは一度に作らなくても、順番に作ることが出来ます。まず

```
pos_list = []
```

とします。[] は空のリストです。

```
pos_list.append([200, 0])
```

とすれば、pos_list は [[200, 0]] となります。.append(要素) はメソッドと呼ばれるリストに付随した関数で、指定した要素をリストの最後に追加します。さらに

```
pos_list.append([200, 200])
```

とすれば、pos_list は [[200, 0], [200, 200]] となります。さらに、

```
pos_list.append([0, 200])
pos_list.append([0, 0])
```

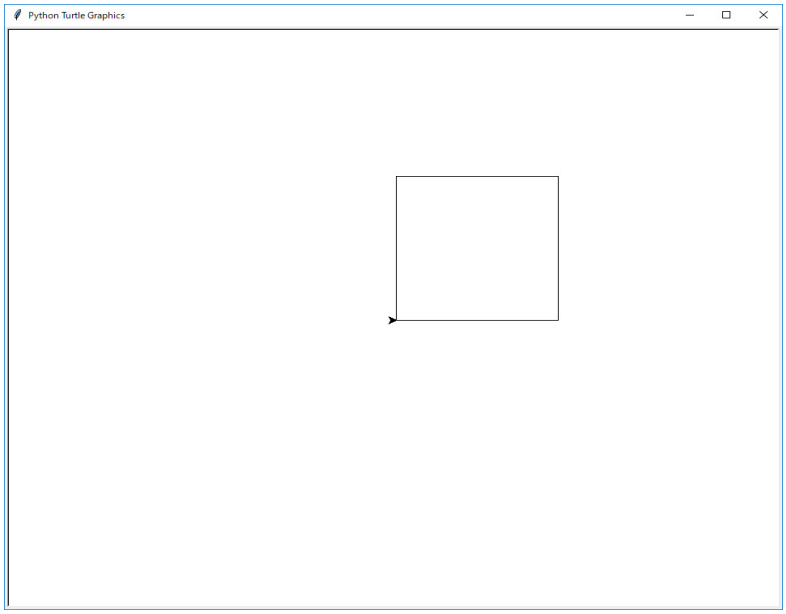
とすれば、pos_list は [[200, 0], [200, 200], [0, 200], [0, 0]] となります。最後に

```
for x in pos_list:
    setpos(x)
```

とすれば、一辺２００の正方形を描きます。即ち、

```
from turtle import *
reset()
pos_list = []
pos_list.append([200, 0])
pos_list.append([200, 200])
pos_list.append([0, 200])
pos_list.append([0, 0])
for x in pos_list:
    setpos(x)
```

のプログラムを実行すれば



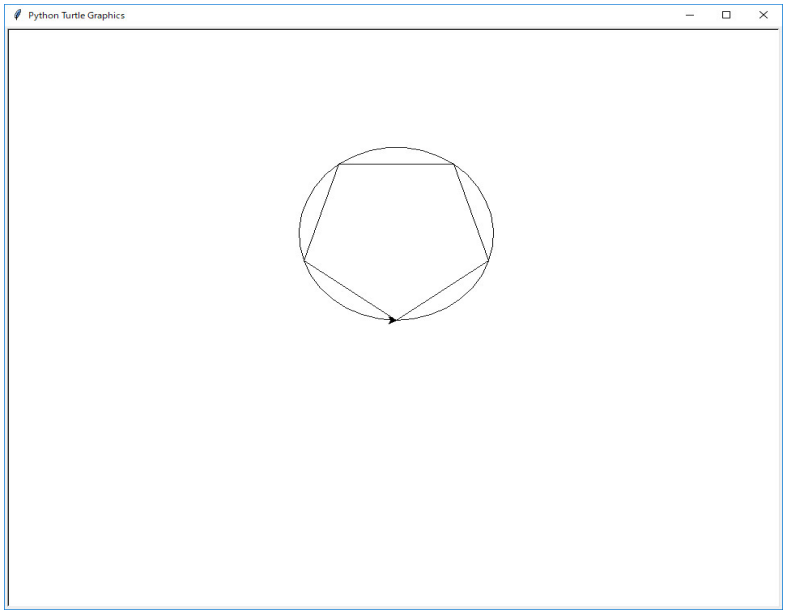
を描きます。

タートルの現在の座標は position() という関数（省略形は pos()）で知ることが出来ます。タートルの現在位置を $(0.00, 0.00)$ のようなタプルで返します。れを覚えていて、円に内接する図形を描くときに使います。

つまり、

```
reset()
pos_list = []
for i in range(5):
    circle(120, 360/5)
    pos_list.append(position())
for x in pos_list:
    setpos(x)
```

を実行すれば、



円と円に内接する正 5 角形を描きます。

```
pos_list = []
for i in range(5):
    circle(120, 360/5)
    pos_list.append(position())
```

で、5 分の 1 の円弧を描きながら、その座標を pos_list に追加していって、

```
for x in pos_list:
    setpos(x)
```

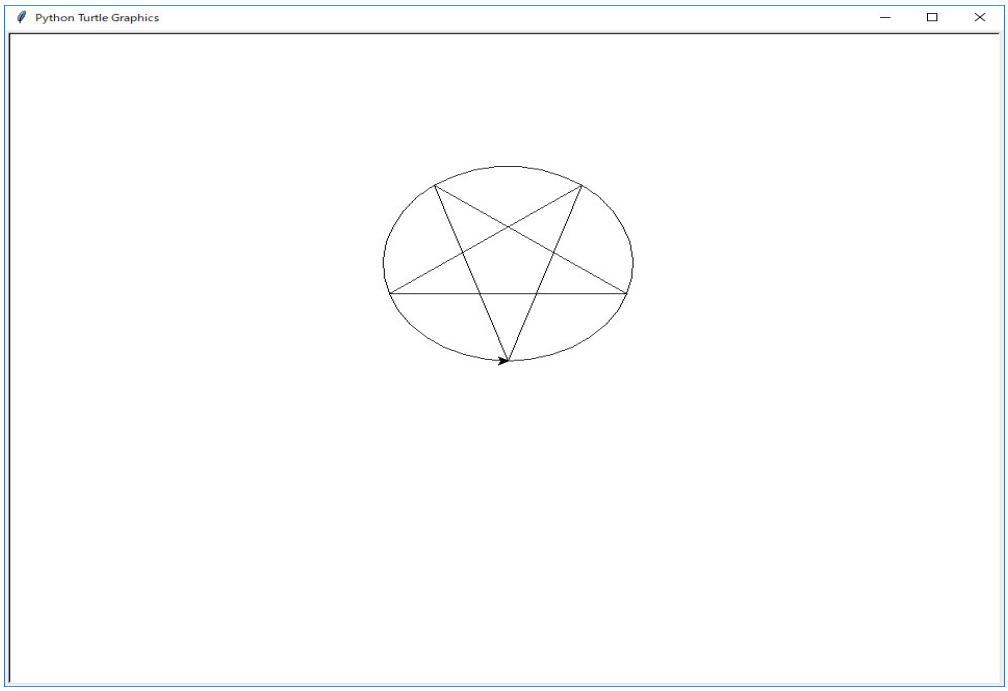
で、 pos_list から保存した座標を順番に取り出し、正五角形を描いています。
　後半の部分は

```
for x in range(5):
    setpos(pos_list[x])
```

としても良いです。pos_list[x]　で pos_list の x 番目の要素を取り出すことが出来ます。リストやタプルの要素は 0 から 0，1，2，3、… と数えます。
　また

```
from turtle import *
reset()
pos_list = []
for i in range(5):
    circle(120, 360/5)
    pos_list.append(position())
pos_list = pos_list * 2
for i in range(1, len(pos_list), 2):
    setpos(pos_list[i])
```

を実行すれば、円に内接した星形正 5 角形を描きます。



ここで

```
pos_list = pos_list * 2
```

とすれば pos_list は　pos_list を二個連結したリストになります。

```
for i in range(1, len(pos_list), 2):
    setpos(pos_list[i])
```

　の部分で使っている range(1, len(pos_list), 2) は 1 以上 len(pos_list) 未満の奇数のリストです。最初の 1 は 1 から始まり、最後の 2 が増分で、2 づつ増やしながら len(pos_list) 未満の数のリストを作る関数です。len(pos_list) は pos_list の要素の個数です。
　円に内接した星形正 5 角形を描くには、

```
pos_list = pos_list * 2
```

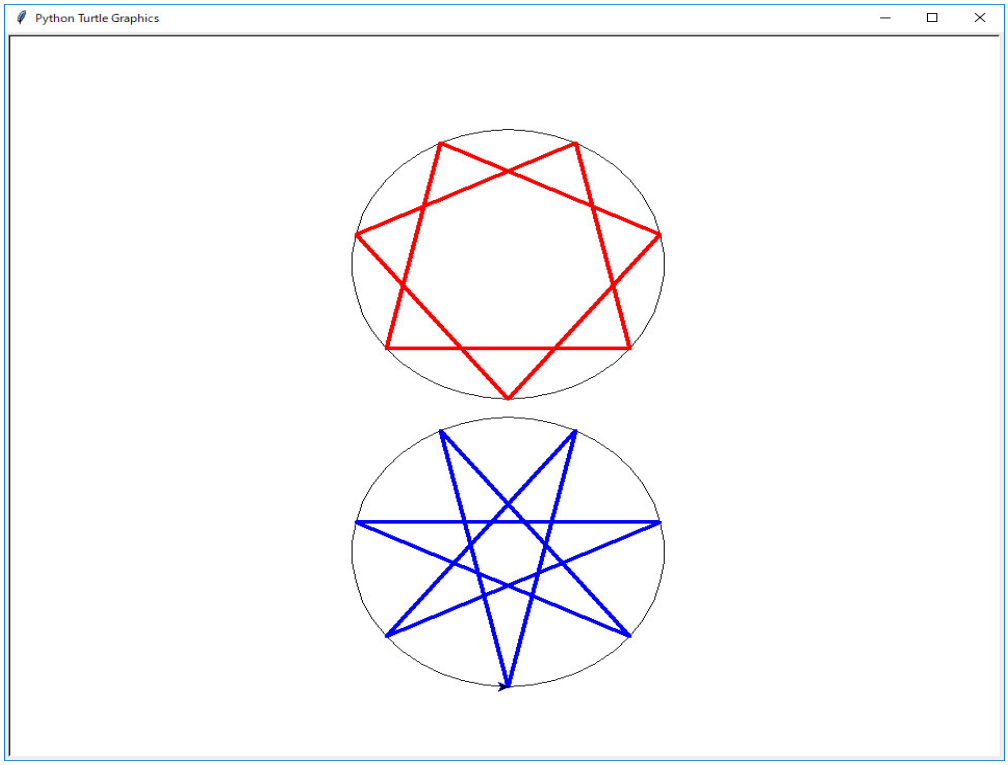
として、リストを連結しなくても、この部分以下を

```
for i in range(5):
    setpos(pos_list[(2*i+1) % 5])
```

としても良かったです。 % は余りを計算する演算子です。
　問題：
　円に内接する二種類の星形正 7 角形を描くプログラムを作りなさい。
　解答例は

```
from turtle import *
reset()
pos_list = []
for i in range(7):
    circle(150, 360/7)
    pos_list.append(position())
pensize(4)
pencolor('red')
for i in range(7):
    setpos(pos_list[(2*i+1)%7])
pos_list = []
pu()
setpos(0,-320)
pd()
pensize(1)
pencolor('black')
for i in range(7):
    circle(150, 360/7)
    pos_list.append(position())
pensize(4)
pencolor('blue')
for i in range(7):
    setpos(pos_list[(3*i+2)%7])
```

です。実行すると

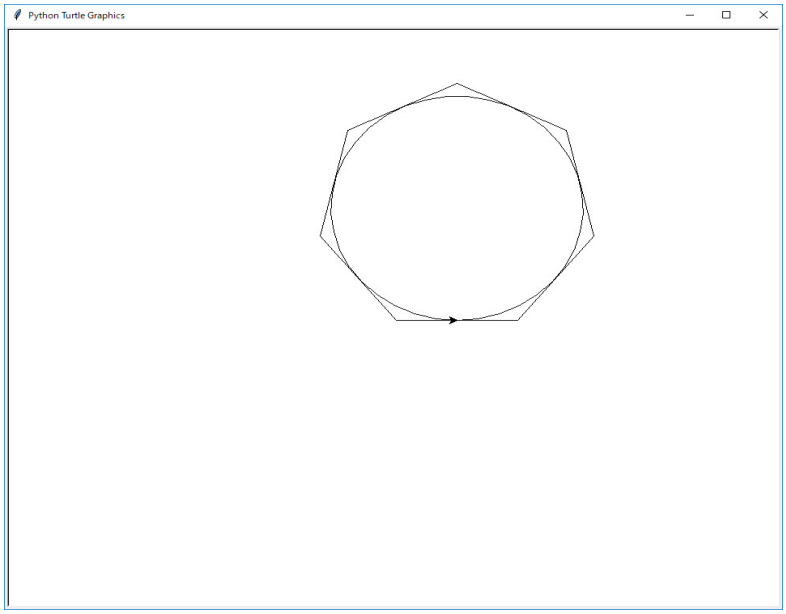


を描きます。

正 n 角形に内接する円を描くプログラムは、三角関数を使って

```
from turtle import *
reset()
from math import *
n = 7
a = 150
for i in range(n):
    fd(a)
    lt(360.0/n)
fd(a/2)
r = a/2.0/tan(pi/n)
circle(r)
```

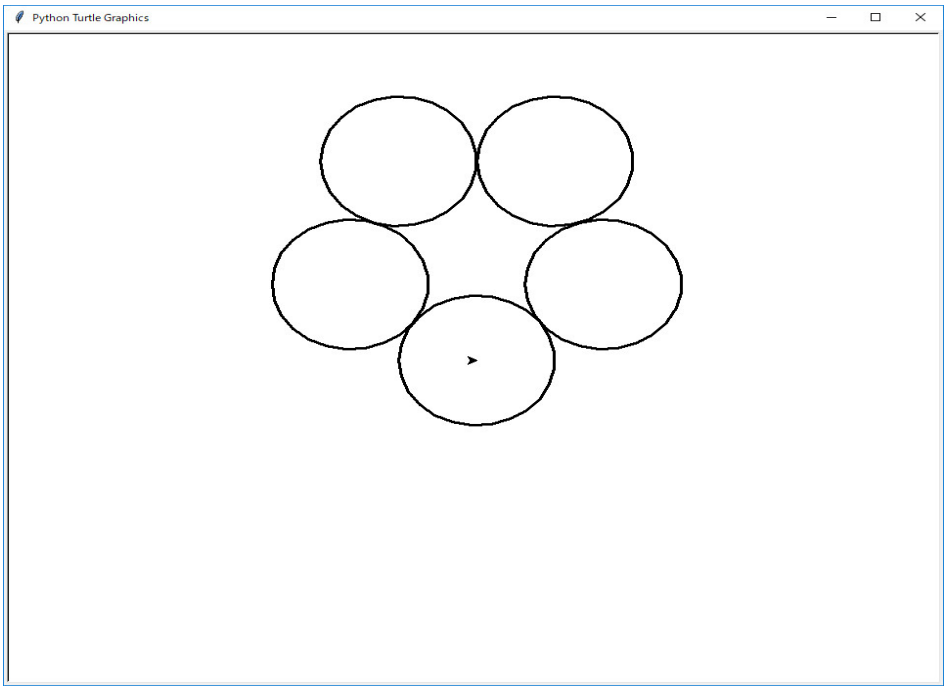
とすれば良いです。a が一辺の長さです。実行すると



を描きます。**注意：三角関数の引数はラジアンで与えなければなりません。**

練習問題：

1. 次のような図を描くプログラムを作れ。



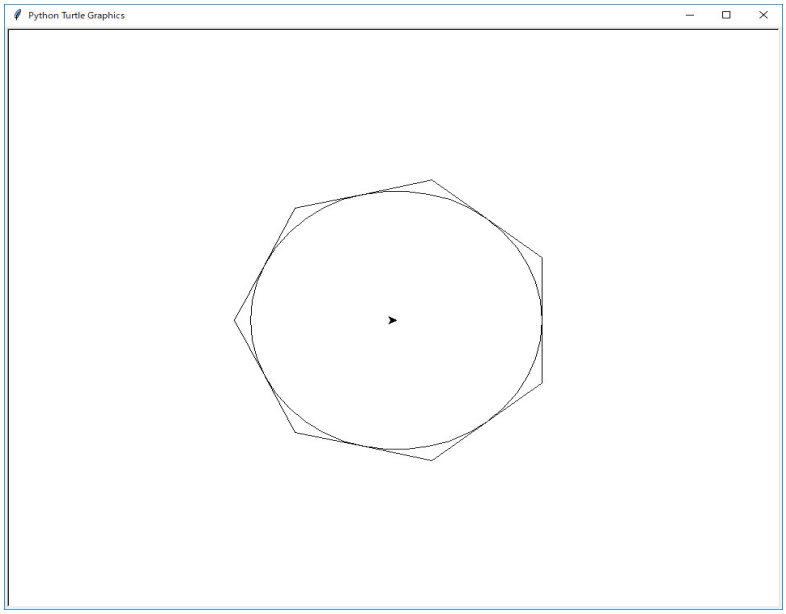
2. 円に外接する正 n 角形を描くプログラムを作れ。
3. 円に外接する星形正 n 角形を描くプログラムを作れ。
4. 星形正 n 角形に内接する円を描くプログラムを作れ。

問題 1 の解答例

```
from turtle import *
reset()
def en(r):
    pu()
    fd(r)
    lt(90)
    pd()
    circle(r)
    pu()
    rt(90)
    bk(r)
lt(36)
r=80
pensize(3)
for i in range(5):
    en(r)
    pu()
    fd(2*r)
    pd()
    lt(72)
rt(36)
```

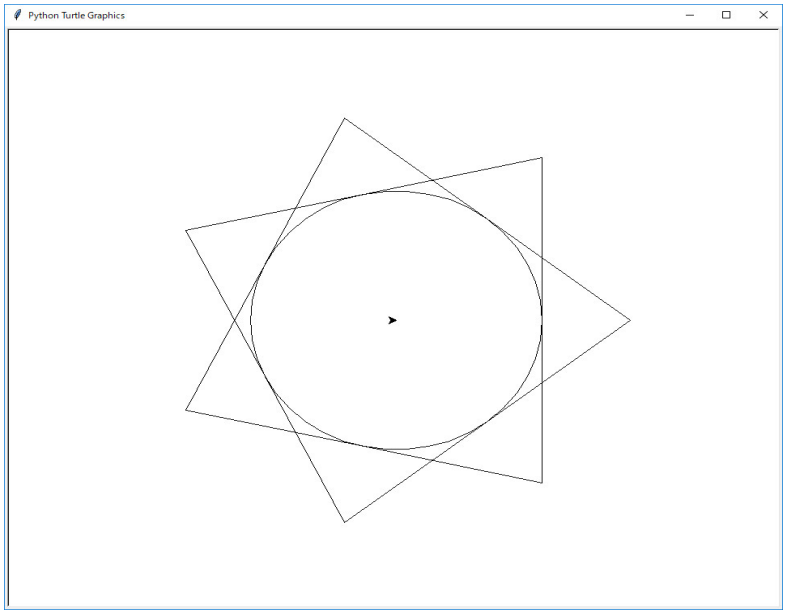
問題 2 の解答例

```
from turtle import *
from math import *
r = 180
n = 7
x = r*tan(pi/n)
pu()
fd(r)
lt(90)
pd()
circle(r)
for i in range(n):
    fd(x)
    lt(360/n)
    fd(x)
rt(90)
pu()
bk(r)
pd()
```



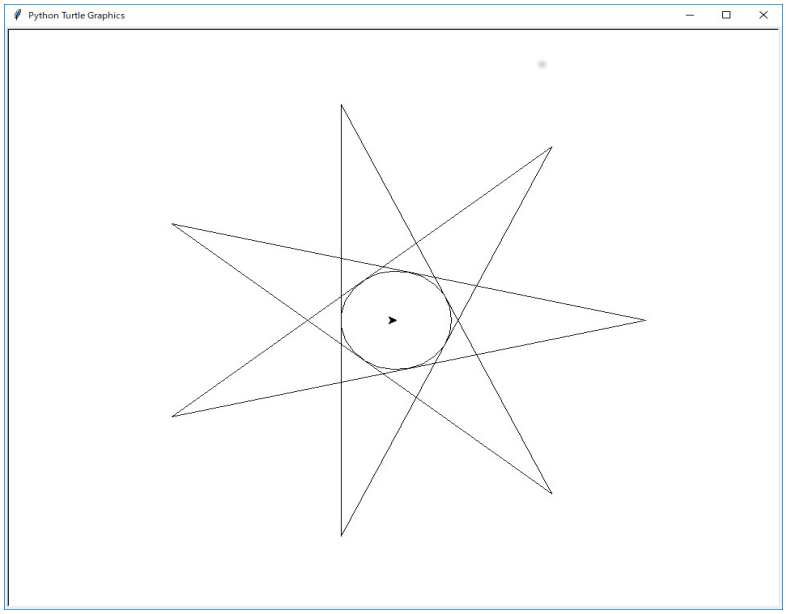
問題３の解答例

```
from turtle import *
from math import *
r = 180
n = 7
s = 2
x = r*tan(s*pi/n)
pu()
fd(r)
lt(90)
pd()
circle(r)
for i in range(n):
    fd(x)
    lt(360*s/n)
    fd(x)
rt(90)
pu()
bk(r)
pd()
```



問題４の解答例

```
from turtle import *
from math import *
a = 600
n = 7
s = 3
r = a/2/tan(s*pi/n)
y = r/cos(s*pi/n)
pu()
fd(y)
lt(90+180*s/n)
pd()
for i in range(n):
    fd(a)
    lt(360*s/n)
rt(90+180*s/n)
pu()
bk(y)
fd(r)
lt(90)
pd()
circle(r)
pu()
rt(90)
bk(r)
pd()
```

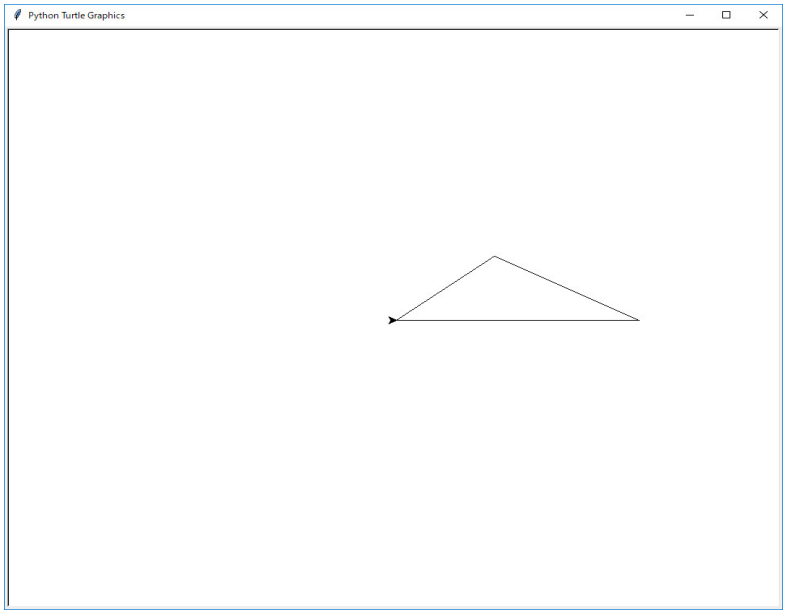


練習問題：

1. 三角形の三辺 a, b, c が与えられた時、三角形を描くプログラムを作れ。
2. 三角形の二辺 a, b と夾角 C が与えられた時、三角形を描くプログラムを作れ。
3. 三角形の二角 B, C と夾辺 a が与えられた時、三角形を描くプログラムを作れ。

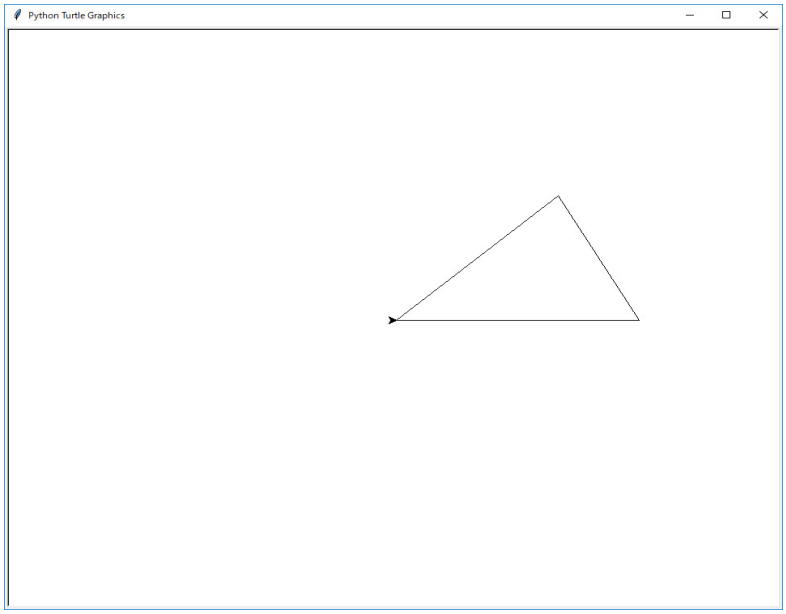
問題 1 の解答例： これは三角形の余弦定理を使えば良いです。

```
from turtle import *
from math import *
a = 300
b = 200
c = 150
A = acos((b*b+c*c-a*a)/(2*b*c))/pi*180
B = acos((c*c+a*a-b*b)/(2*c*a))/pi*180
C = acos((a*a+b*b-c*c)/(2*a*b))/pi*180
fd(a)
lt(180-C)
fd(b)
lt(180-A)
fd(c)
lt(180-B)
```



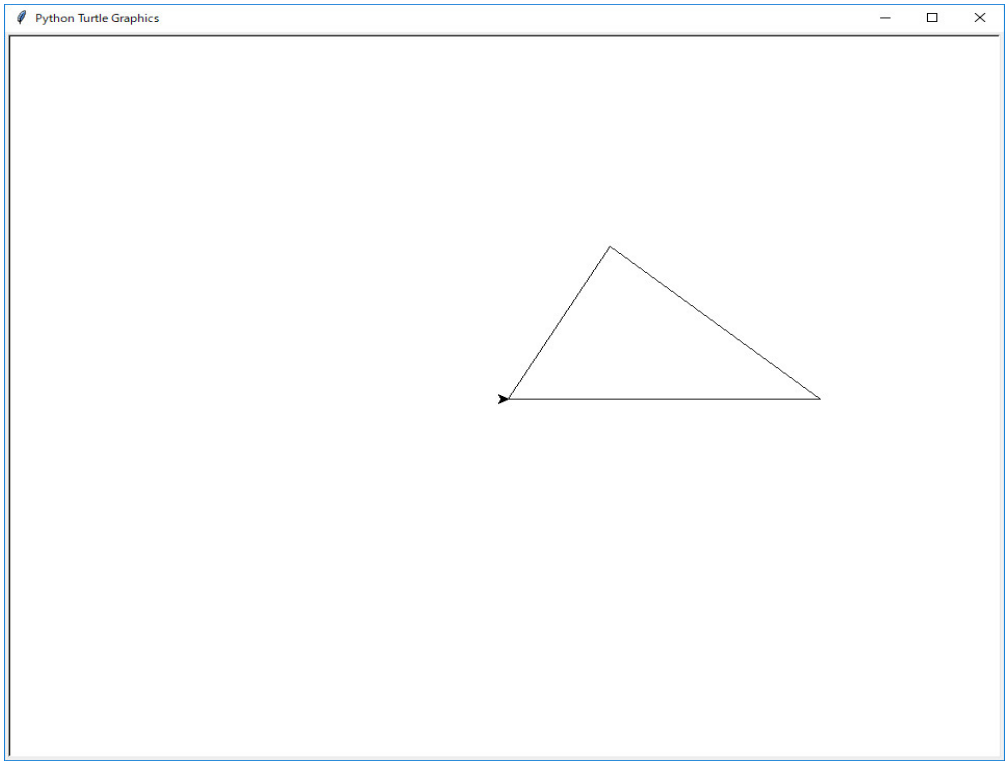
問題２の解答例：　これは出発点の座標を position() 命令を使って覚えて置き、setpos() 命令で復元すればよいです。

```
from turtle import *
from math import *
from turtle import *
from math import *
a = 300
b = 200
C = 60
z = position()
fd(a)
lt(180-C)
fd(b)
setpos(z)
rt(180-C)
```



問題３の解答例：　これは三角形の正弦定理を使えば良いです。

```
from turtle import *
from math import *
a = 300
B = 60
C = 40
A = 180-B-C
rA = A*pi/180
rB = B*pi/180
rC = C*pi/180
b = a*sin(rB)/sin(rA)
c = a*sin(rC)/sin(rA)
fd(a)
lt(180-C)
fd(b)
lt(180-A)
fd(c)
lt(180-B)
```



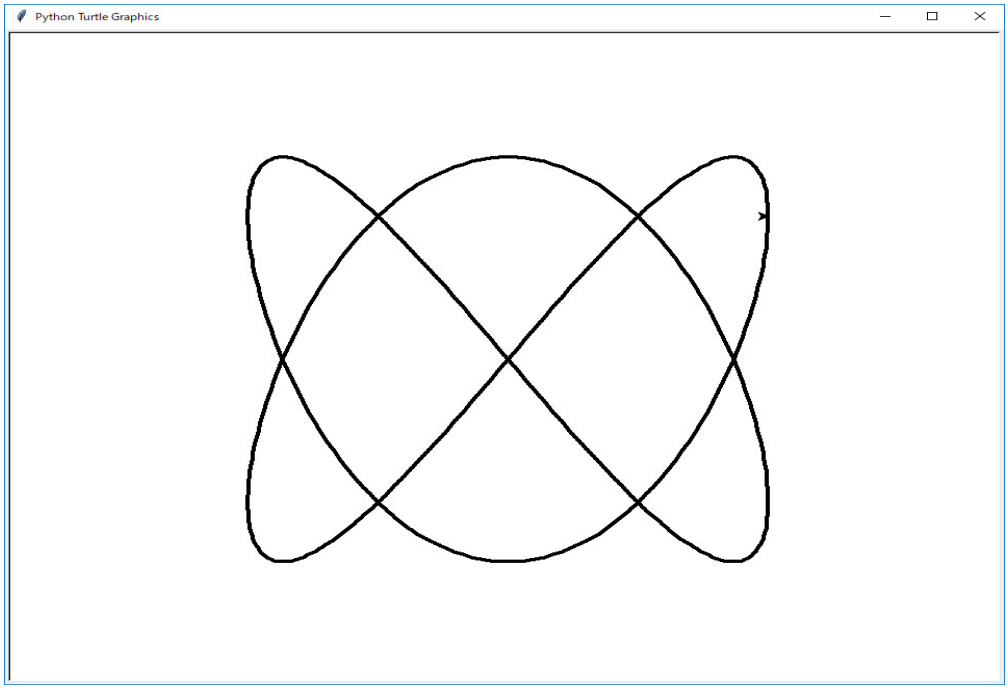
setpos() を使うと普通のグラフを描くことも出来ます。リサージュ曲線を描いてみましょう。リサージュ曲線は媒介変数 t を使って

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

で表される曲線です。

```
from turtle import *
from math import *
A = 250
B = 250
a = 2.0
b = 3.0
delta = pi/4
l = [(A*cos(a*t*pi/180),B*sin(b*t*pi/180+delta)) for t in range(0, 361)]
reset()
pu()
pensize(4)
setpos(l[0])
pd()
for x in l:
    setpos(x)
```

を実行すると

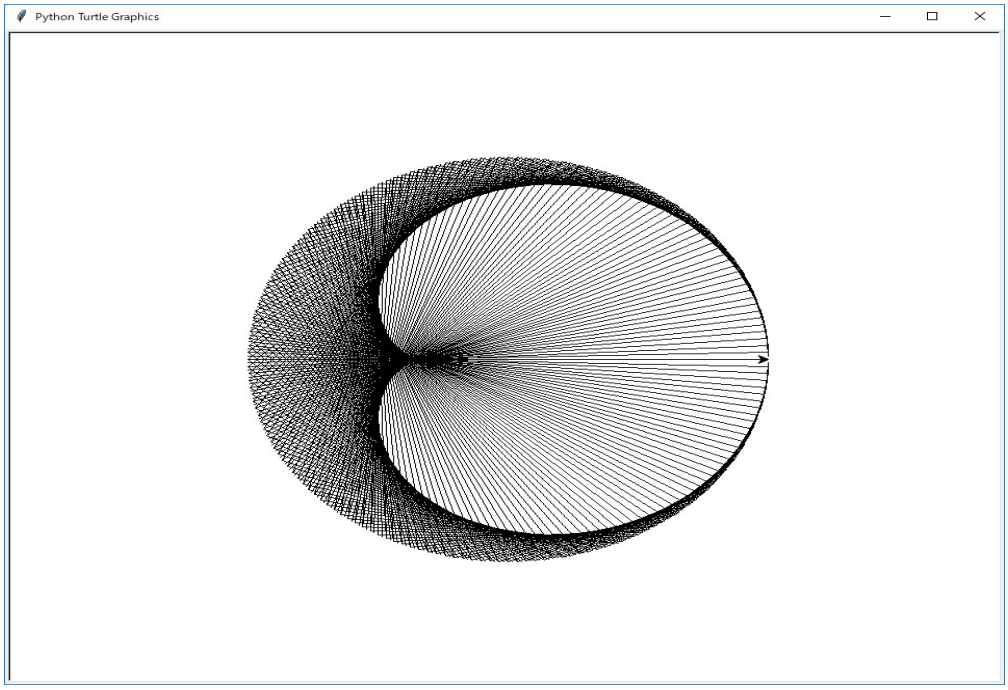


を描きます。

円周上の点 $(A\cos(t), A\sin(t))$ と $(A\cos(2t), A\sin(2t))$ を結ぶ線分群を描くプログラムは次のようになります。

```
from turtle import *
from math import *
A = 250
l = [[(A*cos(t*pi/180),A*sin(t*pi/180)),
      (A*cos(2*t*pi/180),A*sin(2*t*pi/180))] for t in range(0, 361)]
reset()
for x in l:
    pu()
    setpos(x[0])
    pd()
    setpos(x[1])
```

これを実行すると

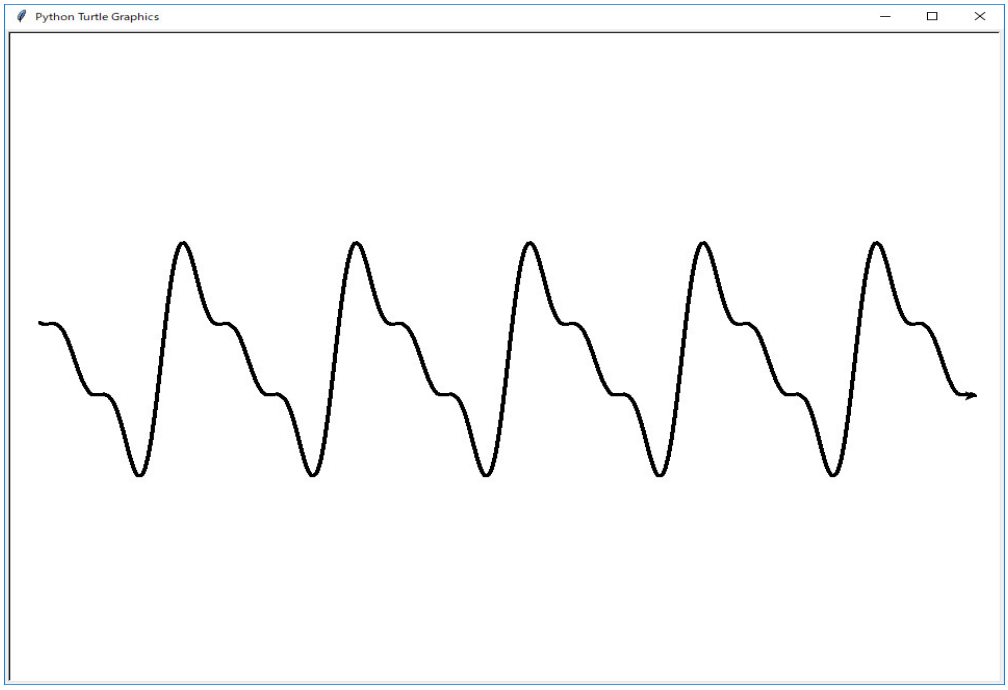


を描きます。

関数 $y = \sin(x) + \sin(2x)/2 + \sin(3x)/3$ を描くプログラムは次のようになります。

```
from turtle import *
from math import *
def f(t):
    return sin(t)+sin(2*t)/2+sin(3*t)/3
A = 250
l = [(x, 100*f(3*pi*x/250)) for x in range(-450, 450)]
reset()
pu()
pensize(4)
setpos(l[0])
pd()
for p in l:
    setpos(p)
```

これを実行すると

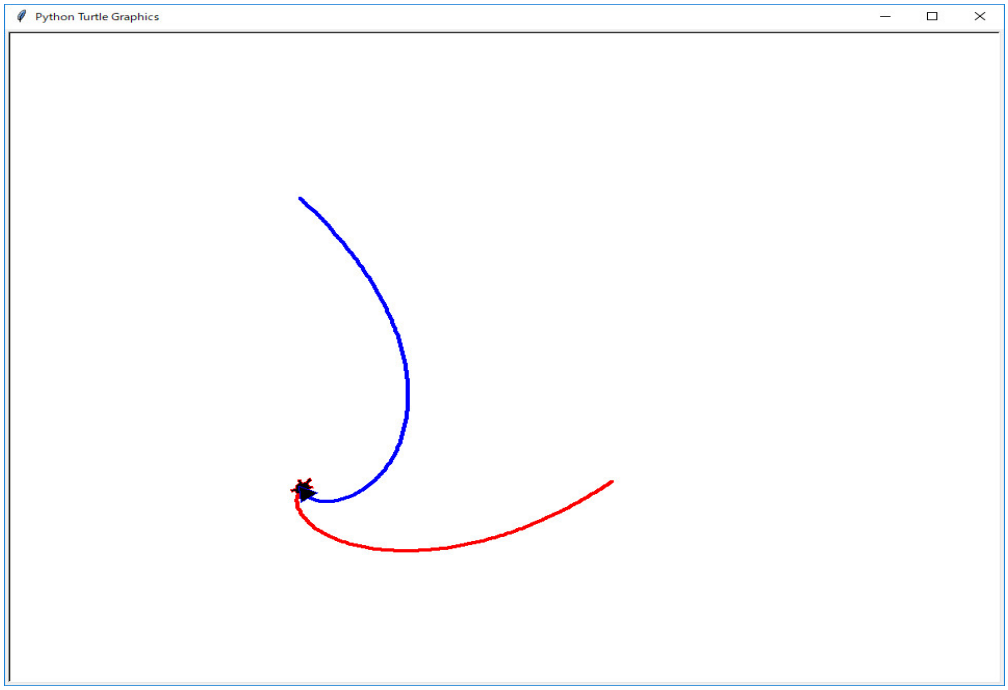


を描きます。

動物の動きをシミュレートしてみましょう。

```
from turtle import *
ht()
p = Turtle()
p.shape("triangle")
p.pensize(4)
p.pencolor('blue')
t = Turtle()
t.shape("turtle")
t.pensize(4)
t.pencolor('red')
p.pu()
p.setpos(-200, 200)
p.pd()
p.lt(90)
t.pu()
t.setpos(100,-150)
t.pd()
while p.distance(t) > 8:
    t.setheading(t.towards(p)+90)
    t.fd(4)
    p.setheading(p.towards(t))
    p.fd(5)
```

とすると



を描きます。

```
ht()
```

は hide turtle の略で、デフォルトのタートルを見えなくしています。見えるようにするには

```
st()
```

とします。show turtle の略です。

この例題では、二つのタートルを使いたいので、デフォルトのタートルを消して、二つのタートルを生成します。

```
p = Turtle()
```

は、新しいタートルを生成し、p に代入しています。

```
p.shape("triangle")
```

は、そのタートルの形を三角形にしています。タートル p に対する命令は p. に続けて書きます。

同様に

```
t = Turtle()
t.shape("turtle")
```

は、新しいタートル t を生成し、そのタートルの形を亀にしています。

```
while p.distance(t) > 8:
    t.setheading(t.towards(p)+90)
    t.fd(4)
    p.setheading(p.towards(t))
    p.fd(5)
```

の p.distance(t) の distance() は距離を与える関数で

```
turtle.distance(x, y=None)

Parameters:
^^e2^^80^^a2x ^^e2^^80^^93 a number or a pair/vector of numbers or a turtle instance
^^e2^^80^^a2y ^^e2^^80^^93 a number if x is a number, else None

Return the distance from the turtle to (x,y), the given vector,
or the given other turtle, in turtle step units.

>>>>>> turtle.home()
>>> turtle.distance(30,40)
50.0
>>> turtle.distance((30,40))
50.0
>>> joe = Turtle()
>>> joe.forward(77)
>>> turtle.distance(joe)
77.0
```

で説明されているように、引数は座標でも、別のタートルでもいいです。

`while p.distance(t) > 8:` は while 文です。

while で指定した条件が、「真」と評価される間は、ブロックが実行されて、ループが繰り返されます。for 文との違いは、カウンタとなる変数を必ずしも必要としない点です。while 文の場合、指定した条件が「偽」と判定されることが、ループの終了条件となります。while 文でも、繰り返して実行されるブロックの部分は、インデントします。今の場合は、タートル p と t の距離が 8 より大きい間は

```
    t.setheading(t.towards(p)+90)
    t.fd(4)
    p.setheading(p.towards(t))
    p.fd(5)
```

が繰り返されます。

`setheading(to_angle)` はタートルを `to_angle` の方向に頭を向けます。

```
turtle.setheading(to_angle)
turtle.seth(to_angle)

Parameters:
to_angle ^^e2^^80^^93 a number (integer or float)

Set the orientation of the turtle to to_angle. Here are some common directions
in degrees:
```

```
standard mode   logo mode
0 - east             0 - north
90 - north          90 - east
180 - west         180 - south
270 - south        270 - west

>>>>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```

towards(x, y) はタートルが座標 (x,y) の方向に向くために setheading() の引数を何度にすれば良いかを返します。引数はベクトルやペアや他のタートルでも良いです。

```
turtle.towards(x, y=None)

Parameters:
^^e2^^80^^a2x ^^e2^^80^^93 a number or a pair/vector of numbers or a turtle instance
^^e2^^80^^a2y ^^e2^^80^^93 a number if x is a number, else None

Return the angle between the line from turtle position to position specified
 by (x,y), the vector or the other turtle. This depends on the turtle's start
orientation which depends on the mode - "standard"/"world" or "logo").
```

従って

```
    t.setheading(t.towards(p)+90)
    p.setheading(p.towards(t))
```

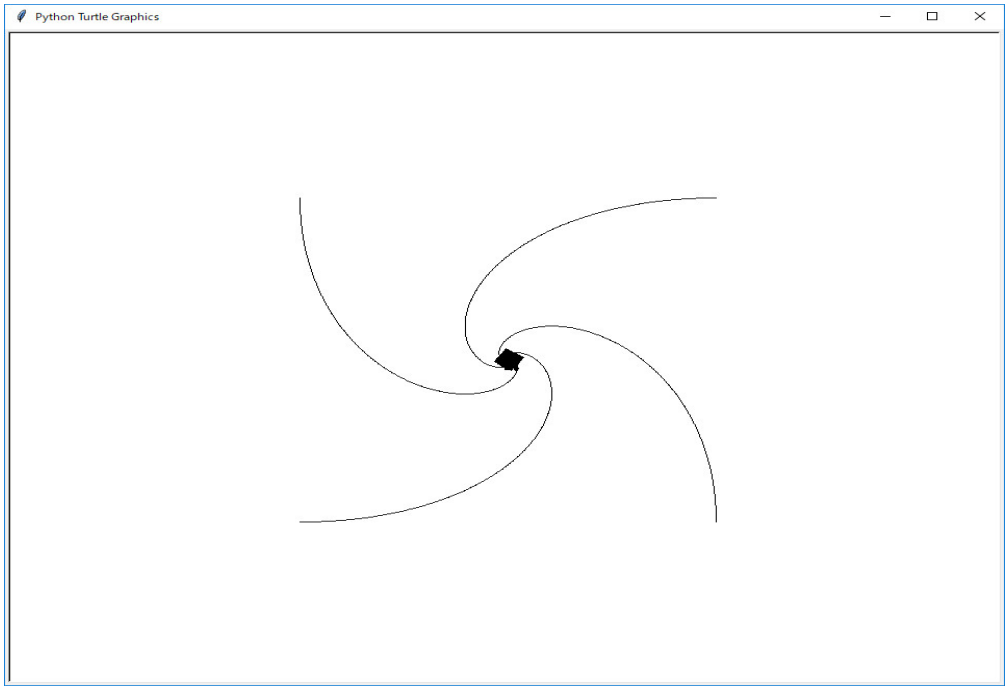
は、それぞれ、タートル t はタートル p の方向に９０度たした方向、タートル p はタートル t の方向に頭を向けます。

次の例題も有名な問題です。正方形の四隅にいるタートルがとれぞれのタートルを追っかけたときに描く曲線を描きます。

```
from turtle import *
ht()
t = [Turtle(),Turtle(),Turtle(),Turtle()]
t[0].shape("triangle")
t[1].shape("turtle")
t[2].shape("circle")
t[3].shape("square")
p = [[-200,200],[-200,-200],[200,-200],[200,200]]
for i in range(4):
    t[i].pu()
    t[i].setpos(p[i])
    t[i].pd()
while t[0].distance(t[1]) > 2:
```

```
    for i in range(4):
        t[i].setheading(t[i].towards(t[(i+1)%4]))
        t[i].fd(1)
```

とすると

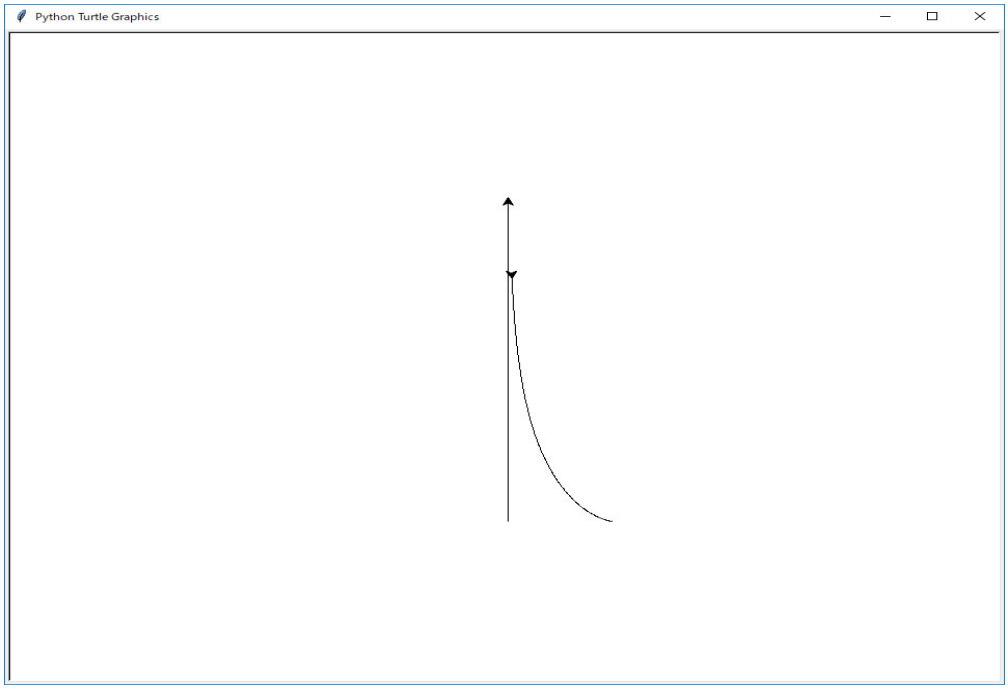


を描きます。

嫌がる犬を引っ張ったとき、犬が描く曲線を tractrix と言います、

```
from turtle import *
reset()
ht()
man = Turtle()
man.lt(90)
man.pu()
man.back(200)
man.pd()
dog = Turtle()
dog.pu()
dog.setpos(100, -200)
dog.pd()
for i in range(400):
    man.fd(1)
    dog.setheading(dog.towards(man)+180)
    dog.back(dog.distance(man)-100)
```

とすると



を描きます。これを回転すると非ユークリッド平面のモデルである偽球になります。この曲線は

$$\frac{dz}{dx} = -\frac{\sqrt{k^2 - x^2}}{x}$$

という微分方程式の解

$$z = k \log \frac{k + \sqrt{k^2 - x^2}}{x} - \sqrt{k^2 - x^2}$$

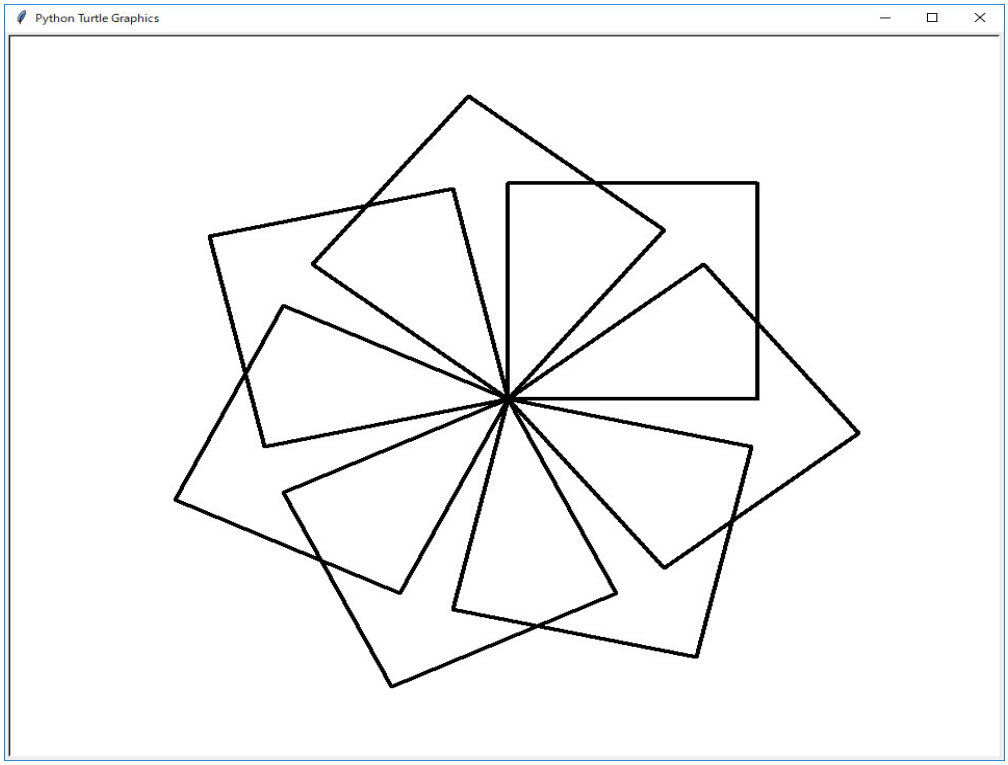
で与えられます。参考書 :「ユークリッド幾何から現代幾何へ」小林昭七著

Python の turtle モジュールにどんな命令があるかはインターネットで調べることが出来ます。

for 文はネストすることが出来、

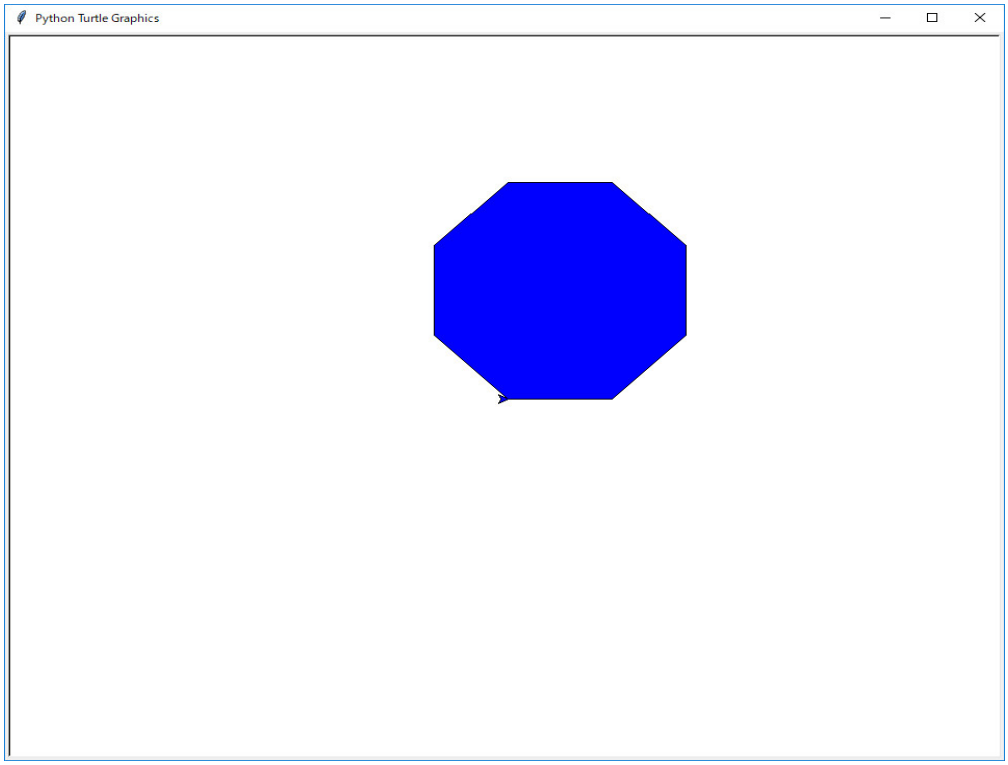
```
from turtle import *
pensize(4)
for i in range(7):
    for j in range(4):
        forward(240)
        left(90)
    left(360.0/7)
```

のように使うことが出来ます。



塗りつぶされた図形を描くには、次のようにします。

```
from turtle import *
fillcolor("blue")
begin_fill()
for i in range(8):
    forward(100)
    left(45)
end_fill()
```




```
fillcolor("blue")
```

は塗りつぶす色を指定します。

```
begin_fill()
.....
end_fill()
```

は `begin_fill()` と `end_fill()` で囲まれた部分で描かれた図形を塗りつぶします。Python2.7 では

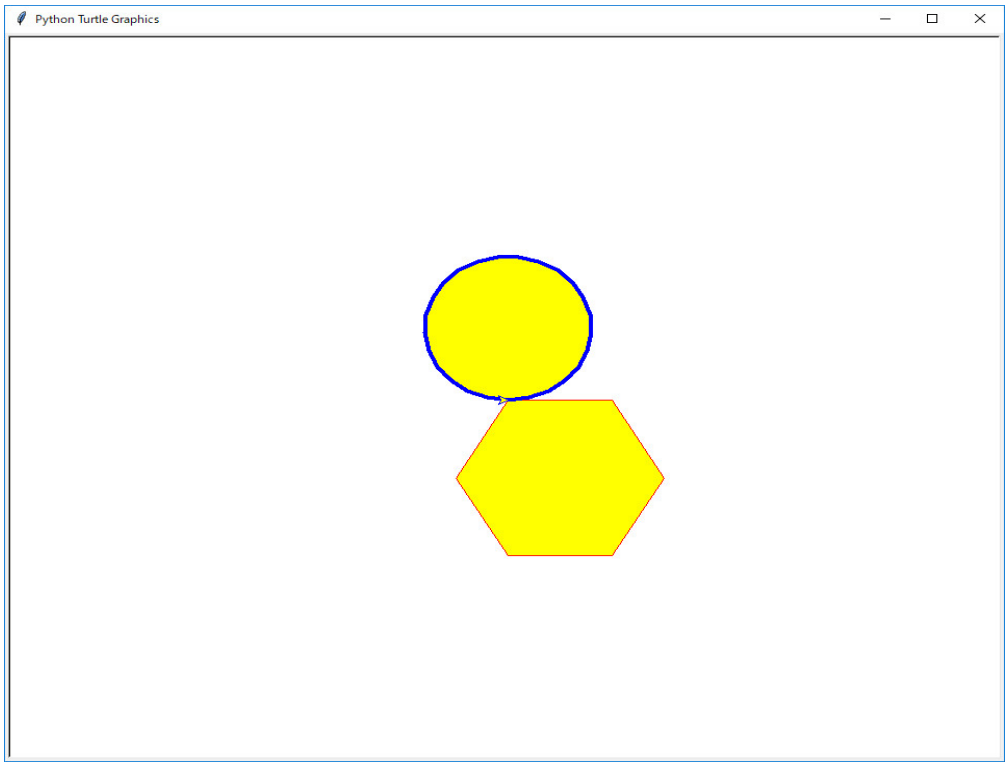
```
fill(True)
.....
fill(false)
```

でも良かったですが、Python3.6 では、廃止されたようです。

```
from turtle import *
reset()
color("red", "yellow")
begin_fill()
for _ in range(6):
    fd (100)
    rt(60)
pencolor("blue")
```

```
pensize(4)
circle(80)
end_fill()
```

とすると次のようになります。




```
from turtle import *
reset()
color("red", "yellow")
begin_fill()
for _ in range(6):
    fd (100)
    rt(60)
pencolor("blue")
pensize(4)
circle(80)
end_fill()
```

の

```
pencolor("blue")
pensize(4)
```

は、ペンの色とペンの大きさを指定しています。

```
color("red", "yellow")
```

は

```
pencolor("red")
fillcolor("yellow")
```

とするのと同じことです。
　どのような色の指定ができるかはインターネットで調べてください。また色の指定は色の名前で指定しなくても

```
pencolor('#F00')
fillcolor('#F0F')
```

や

```
pencolor('#FF0000')
fillcolor('#FF00FF')
```

や

```
pencolor('#FFF000000')
fillcolor('#FFF000FFF')
```

のように光の三原色（赤、緑、青）の割合を示す RGB の数字で指定することも可能です。
　一連の手続きを関数として定義することも出来ます。

```
def square():
    for i in range(4):
        forward(120)
        left(90)
```

とすれば、square() という関数を定義することが出来ます。

```
from turtle import *
def square():
    for i in range(4):
        forward(120)
        left(90)
square():
```

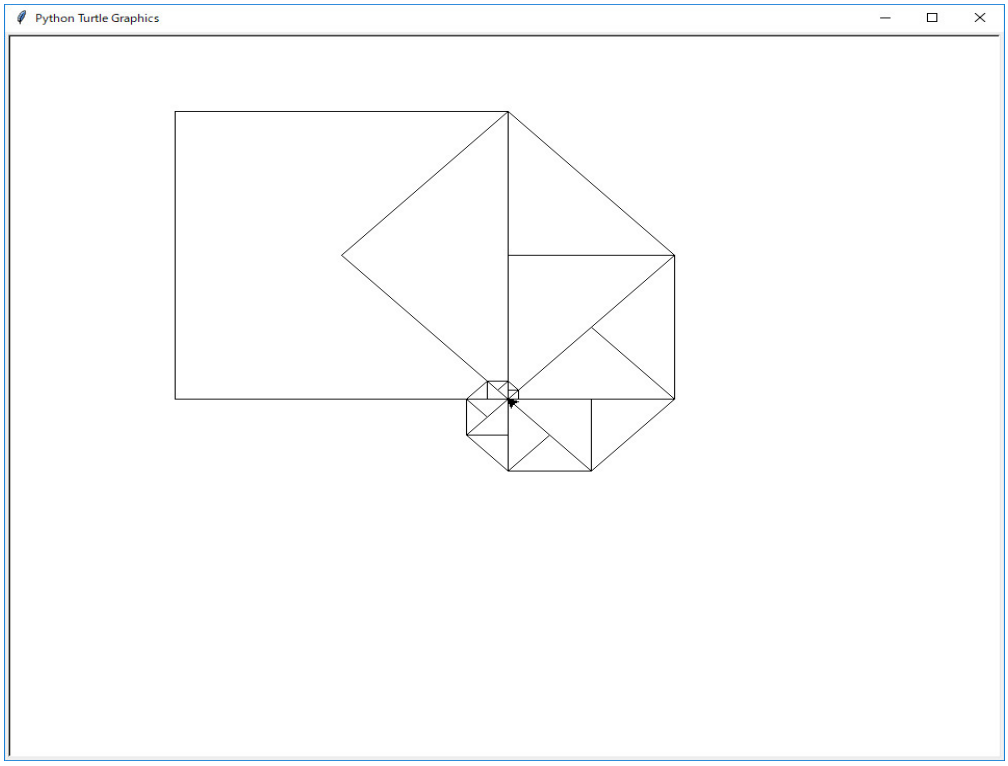
とすれば、正方形を描きます。これでは、いつも一辺１２０の正方形を描きます。

```
def square(size):
    for i in range(4):
        forward(size)
        left(90)
```

とすれば、square(size) の引数 size を色々変えることで、一辺 size の正方形を描けるようになります。

```
from turtle import *
from math import *
def square(size):
    for i in range(4):
        forward(size)
        left(90)
reset()
x = 10
for k in range(11):
    square(x)
    left(45)
    x *= sqrt(2)
```

とすると



こんな絵を描きます。
　まず、最大公約数を計算する関数を

```
def gcd(n, m):
    if m == 0:
        return n
    else:
        return gcd(m, n % m)
```

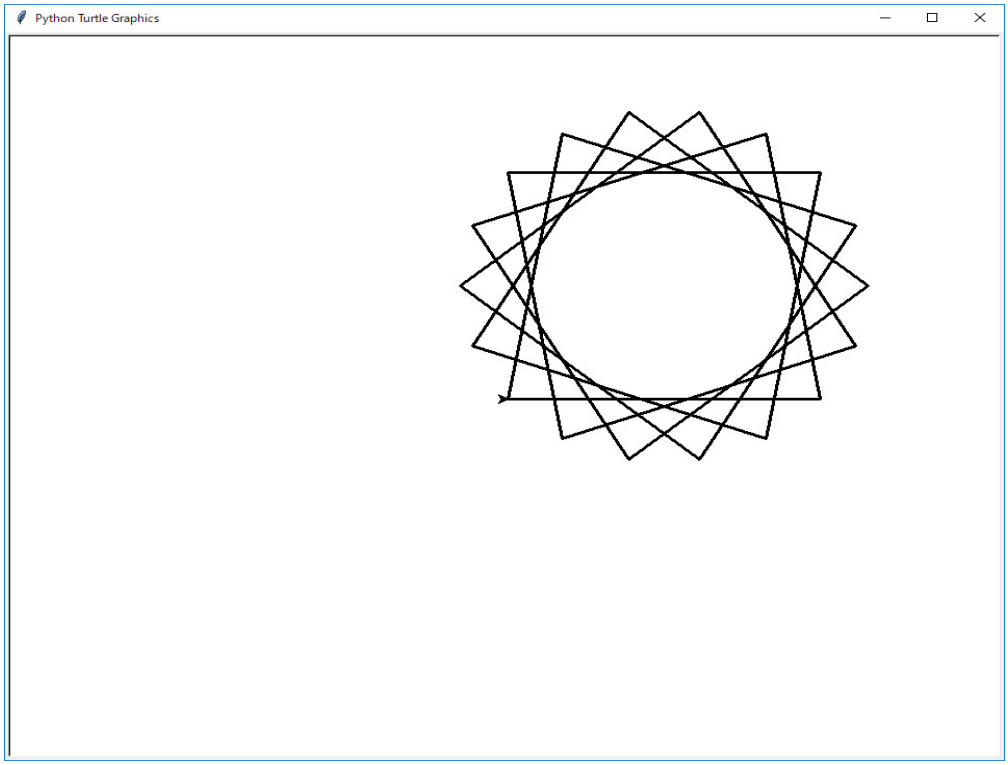
で定義します。次に

```
def poly(size, angle):
    for i in range(360//gcd(360, angle)):
        forward(size)
        left(angle)
```

と定義します。これで外角が angle の多角形を描くことが出来ます。

```
from turtle import *
def gcd(n, m):
    if m == 0:
        return n
    else:
        return gcd(m, n % m)
def poly(size, angle):
    for i in range(360//gcd(360, angle)):
        forward(size)
        left(angle)
pensize(3)
poly(300, 100)
```

とすれば次のような図を描きます。



この poly() は次のように再帰的に定義しても良いです。

```
def poly2(size, angle, total):
```

```
    forward(size)
    left(angle)
    total += angle
    if total % 360 == 0:
        return
    else:
        poly2(size, angle, total)
```

を

```
from turtle import *
def poly2(size, angle, total):
    forward(size)
    left(angle)
    total += angle
    if total % 360 == 0:
        return
    else:
        poly2(size, angle, total)
pensize(3)
poly2(300, 100, 0)
```

とすれば同じ図を描きます。

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。

この条件でプログラムを書くと

```
from turtle import *

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

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
```

となります。

Python では複素数が使えます。

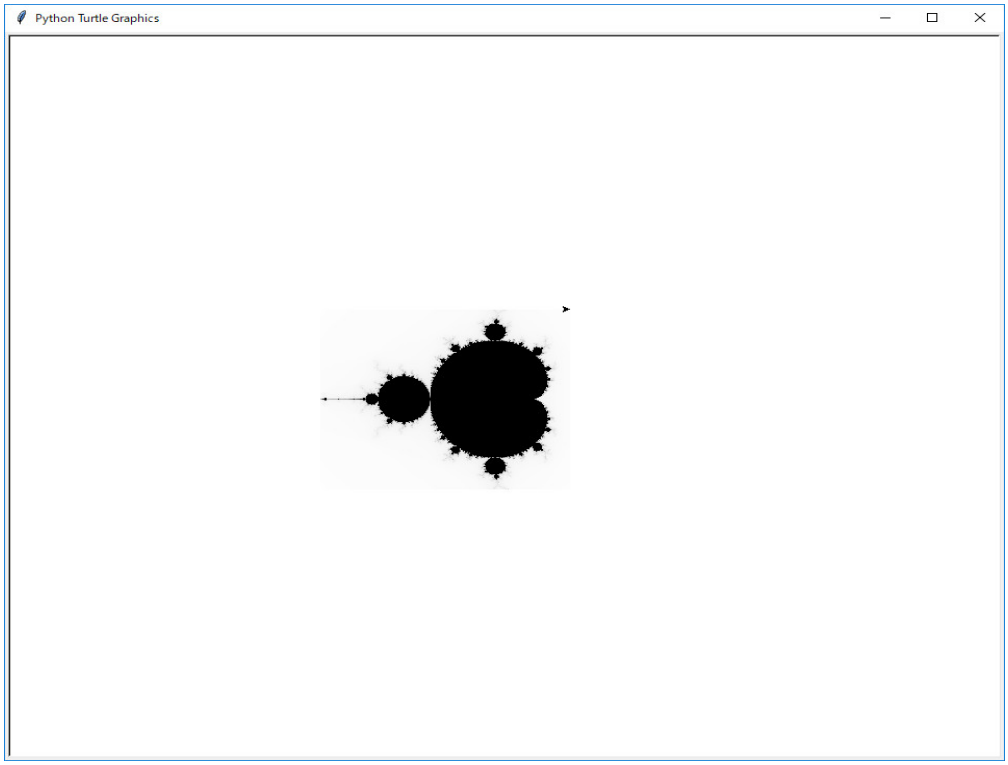
```
    z = 0.0 + 0.0*1j
```

で、z を複素数の変数としています。 $1j$ が数学での i です。

```
    z = complex(0.0,  0.0)
```

でも良いです。

実行すれば

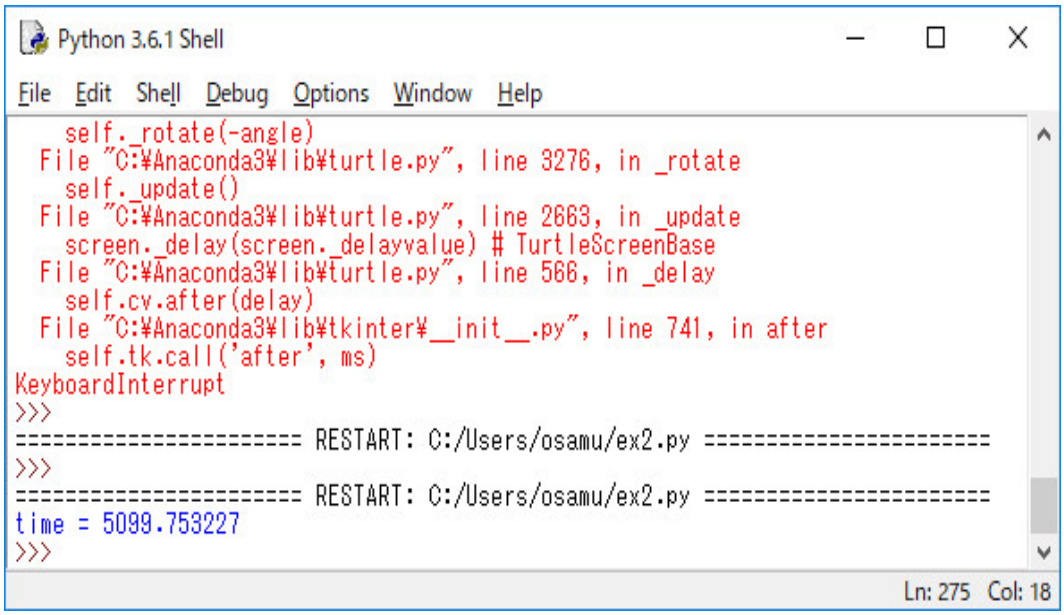


を描きますが、約２時間半かかります。

```
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) )
```

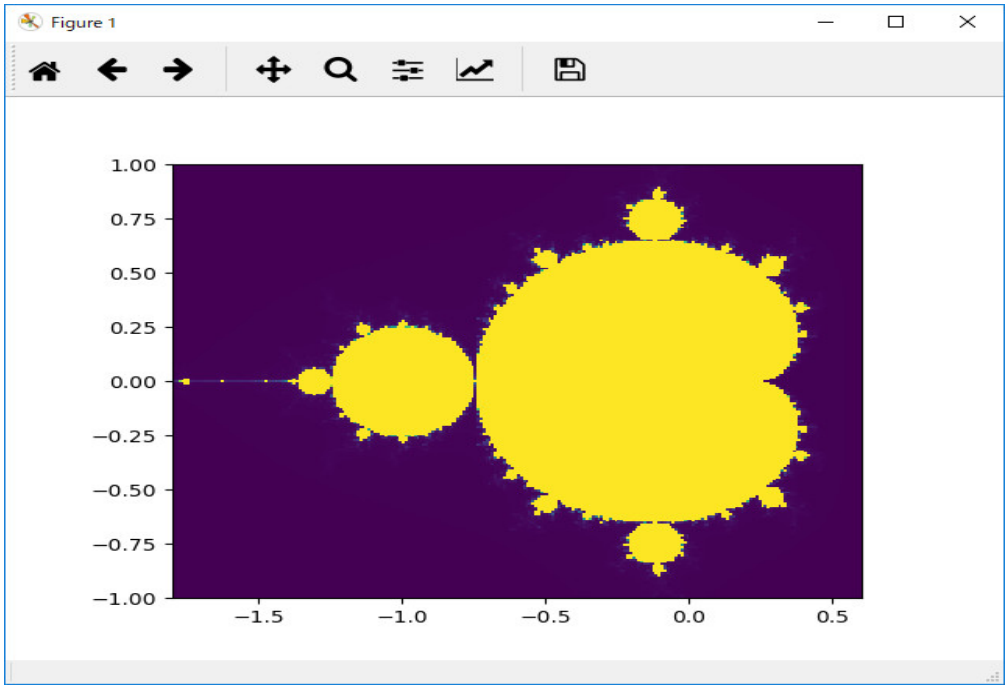
とすれば、時間が測れます。実行すれば



です。ht() でタートルを消すと少し早くなりましたが time = 5099.753227、すなわち、１時間２５分ぐらいです。１９８０年ごろの８ビットのマイコンの時代は、BASIC でプログラムを組み、寝る前にプログラムを動かせば、朝起きたときに図が出来上がっていましたが、今はこらえ性がなくなっています。このような図形を描くことを想定していないタートル・グラフィックスでもこのようにマンデルブロー集合を描くことができますが、図を描くことを Python のライブラリー pylab に任せると実行速度は速くなります。

```
import time
import pylab

def mb(x,y):
    c = complex(x, y)
    z = complex(0.0, 0.0)
    n = 0
    LOOP = 1000

    while (abs(z) < 2 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 = [[0 for i in range(0,len(x))] for j in range(0,len(y))]

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

pylab.imshow(l, extent=(xmin, xmax, ymin, ymax))
end_time = time.time()
print ('time = %f' % (end_time - start_time) )
pylab.show()
```

を実行すると



で、time = 10.595608、すなわち１０秒ぐらいで実行してくれます。描く色の指定はできませんが、「餅は餅屋」で、適切なライブラリーを使うと快適に Python を使うことができます。Python にはたくさんのライブラリーが準備されているので、全体像を知るには学ぶべきことがたくさんあります。やりたいことに応じて、少しづつ学んでいけばいいです。tkinter を使えば、自分の好きな色で素早く描画できます。高速性が高度に要求される囲碁のプログラムなどは C++ でないと作れませんが、構文が簡単で習得が容易な Python を使うのがいいか、なんでも自前でできるが習得の困難なコンパイラの C++ がいいか迷うところです。

タートル・グラフィックスの問題に戻ります。再帰的な定義をすると面白い図を描くことが出来ます。

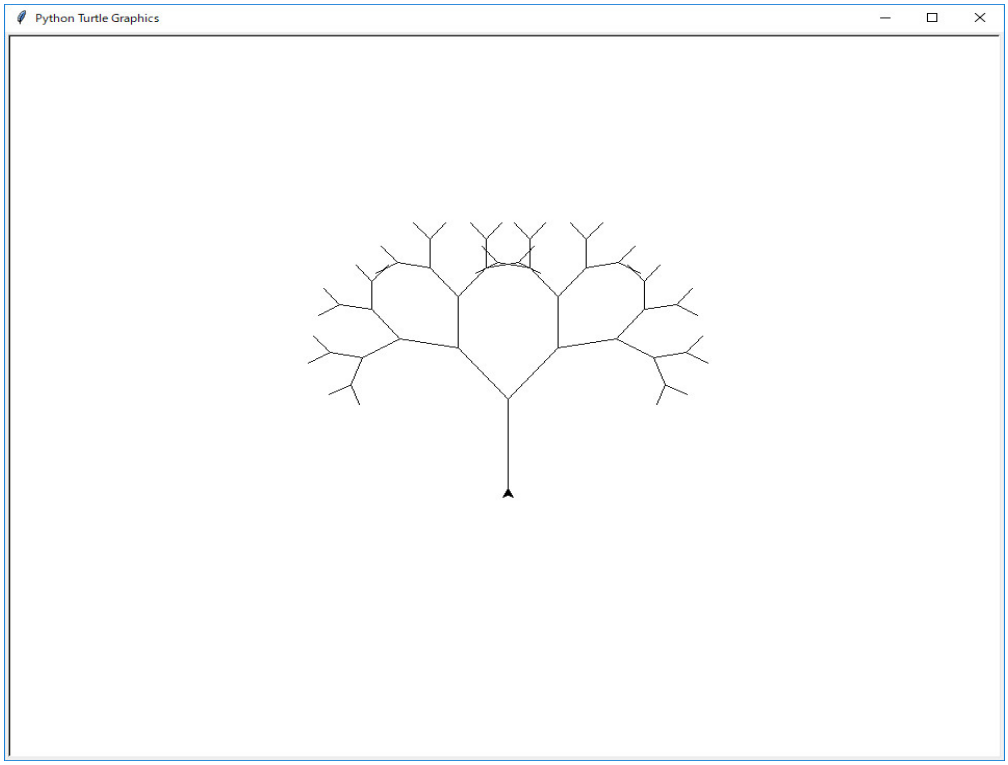
```
def tree(size, angle, depth):
    if depth == 0: return
    fd(size)
    rt(angle)
    tree(size*0.75, angle, depth-1)
    lt(2*angle)
    tree(size*0.75, angle, depth-1)
    rt(angle)
    back(size)
```

と定義して、

```
from turtle import *
def tree(size, angle, depth):
    if depth == 0: return
    fd(size)
```

```
    rt(angle)
    tree(size*0.75, angle, depth-1)
    lt(2*angle)
    tree(size*0.75, angle, depth-1)
    rt(angle)
    back(size)
lt(90)
pu()
back(100)
pd()
tree(100, 40, 6)
```

とすれば、

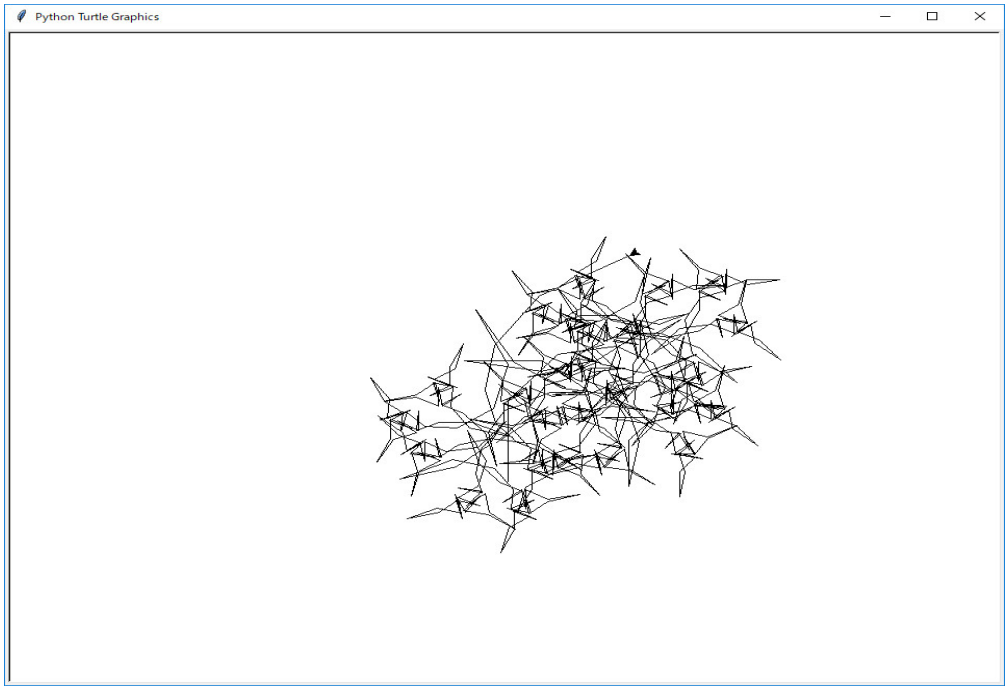


を描きます。

```
from turtle import *
def tree(size, angle1, angle2, angle3, depth):
    if depth == 0: return
    fd(size)
    rt(angle1)
    tree(size*0.75, angle1, angle2, angle3, depth-1)
    lt(angle1-angle2)
    tree(size*0.75, angle1, angle2, angle3, depth-1)
```

```
    lt(angle3-angle2)
    tree(size*0.75, angle1, angle2, angle3, depth-1)
    rt(angle3)
    back(size)

lt(90)
pu()
back(150)
pd()
tree(100, 40, 30, 25, 6)
```

とプログラムして、実行すると

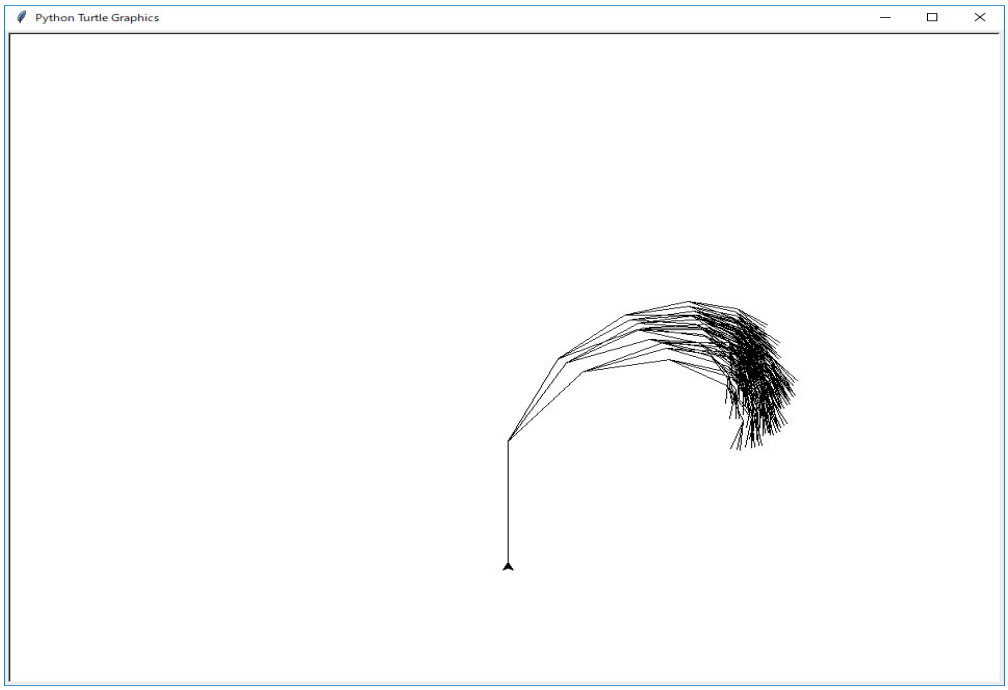


を描きます。このような文法的な誤りではなく、論理的な誤りはコンピュータは指摘してくれません。自分で間違いを見つけて修正しなければなりません。

```
from turtle import *
def tree(size, angle1, angle2, angle3, depth):
    if depth == 0: return
    fd(size)
    rt(angle1)
    tree(size*0.75, angle1, angle2, angle3, depth-1)
    lt(angle1-angle2)
    tree(size*0.75, angle1, angle2, angle3, depth-1)
    lt(angle2-angle3)
    tree(size*0.75, angle1, angle2, angle3, depth-1)
    lt(angle3)
```

```
    back(size)
lt(90)
pu()
back(250)
pd()
tree(150, 40, 30, 25, 6)
```

と修正すれば、

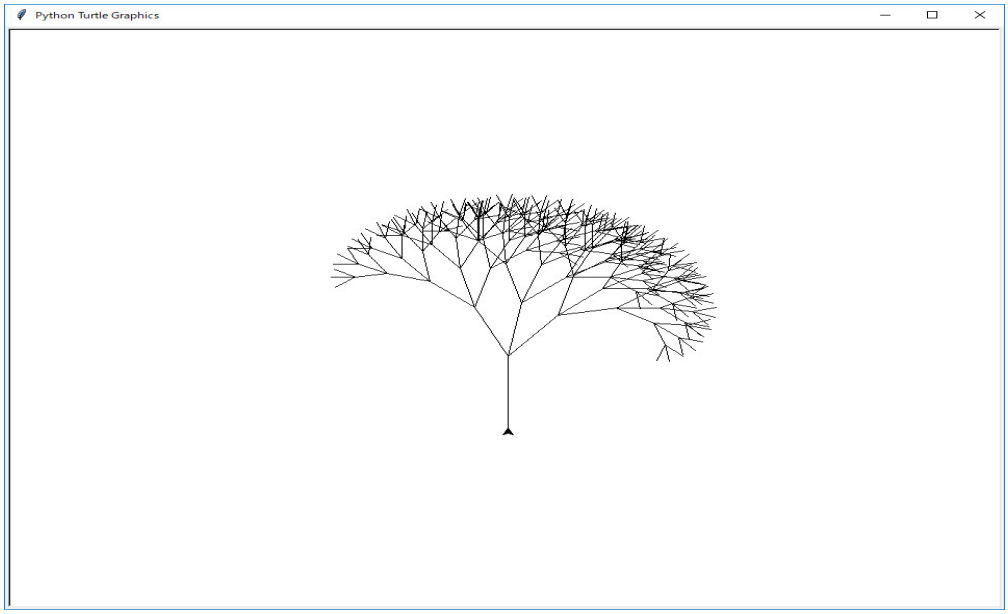


を描きます。

```
from turtle import *
def tree(size, angle1, angle2, angle3, depth):
    if depth == 0: return
    fd(size)
    rt(angle1)
    tree(size*0.75, angle1, angle2, angle3, depth-1)
    lt(angle1-angle2)
    tree(size*0.75, angle1, angle2, angle3, depth-1)
    lt(angle2-angle3)
    tree(size*0.75, angle1, angle2, angle3, depth-1)
    lt(angle3)
    back(size)
lt(90)
pu()
back(150)
pd()
```

```
tree(100, 40, 10, -25, 6)
```

を実行すれば、



を描きます。角度や枝の長さの割合を変えて、いろいろな木を描くことができます。

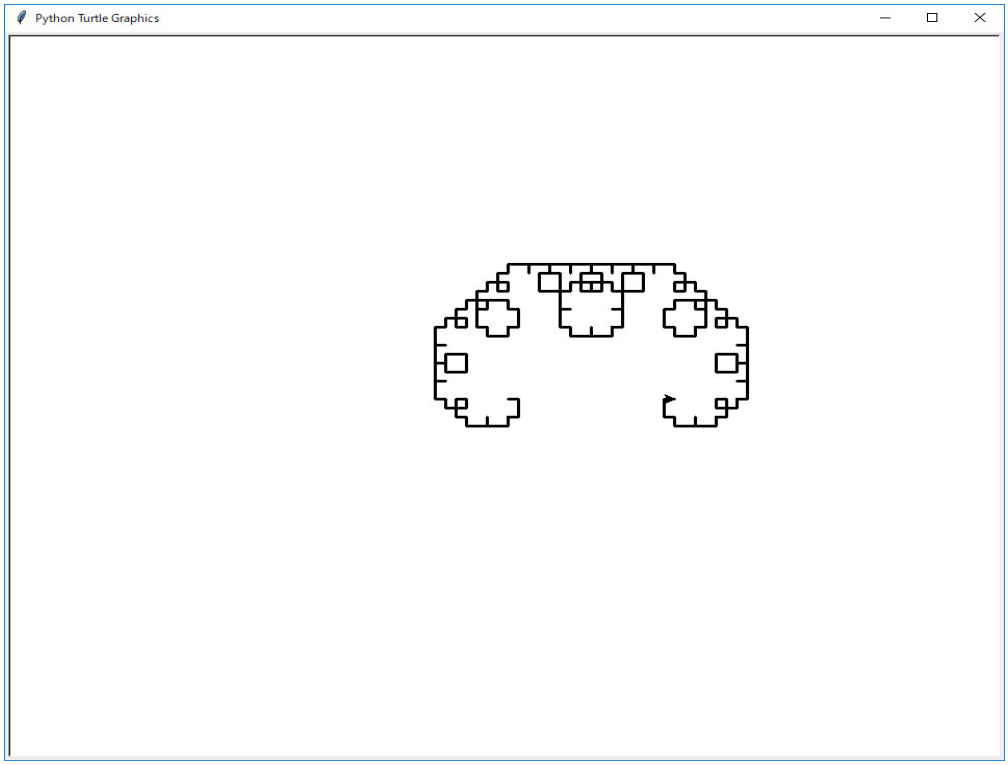
以下では、有名な再帰図形を描いてみます。

```
def c(size, level):
    if level == 0:
        fd(size)
        return
    c(size, level-1)
    rt(90)
    c(size, level-1)
    lt(90)
```

と定義して、

```
from turtle import *
def c(size, level):
    if level == 0:
        fd(size)
        return
    c(size, level-1)
    rt(90)
    c(size, level-1)
    lt(90)
pensize(3)
c(10, 8)
```

とすれば、



を描きます。C 曲線と言います。

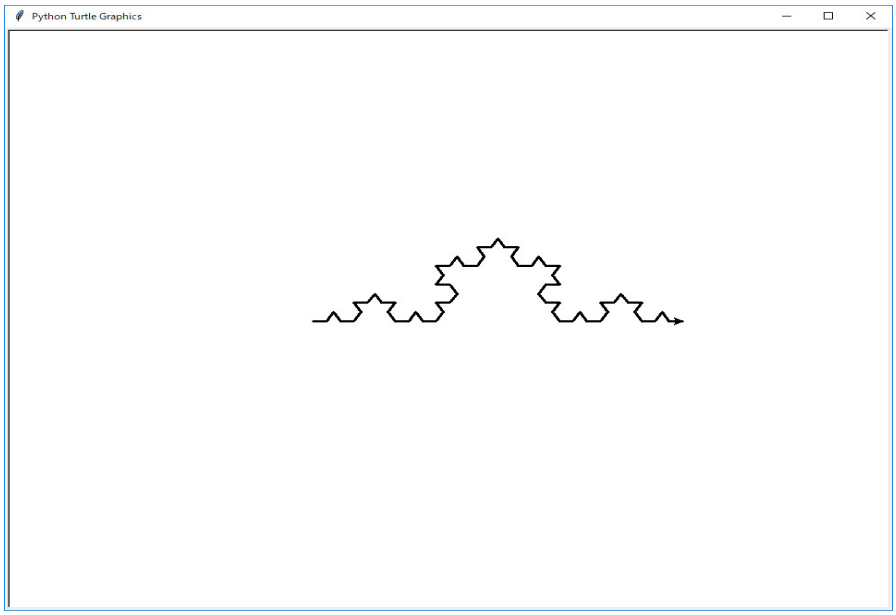
```
def koch(size, level):
    if level == 0:
        fd(size)
        return
    koch(size/3, level-1)
    lt(60)
    koch(size/3, level-1)
    rt(120)
    koch(size/3, level-1)
    lt(60)
    koch(size/3, level-1)
```

と定義して、

```
from turtle import *
def koch(size, level):
    if level == 0:
        fd(size)
        return
    koch(size/3, level-1)
    lt(60)
```

```
    koch(size/3, level-1)
    rt(120)
    koch(size/3, level-1)
    lt(60)
    koch(size/3, level-1)
pensize(3)
pu()
bk(150)
pd()
koch(400, 3)
```

とすれば、

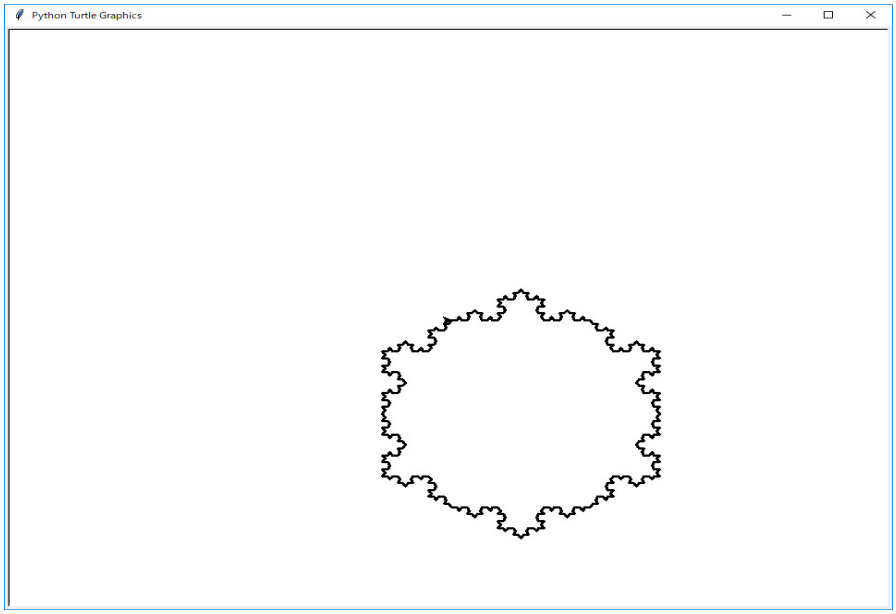


を描きます。koch() が描く曲線をコッホ曲線と言います。

```
from turtle import *
def koch(size, level):
    if level == 0:
        fd(size)
        return
    koch(size/3, level-1)
    lt(60)
    koch(size/3, level-1)
    rt(120)
    koch(size/3, level-1)
    lt(60)
    koch(size/3, level-1)
pensize(3)
```

```
for _ in range(6):
    koch(150, 3)
    rt(60)
```

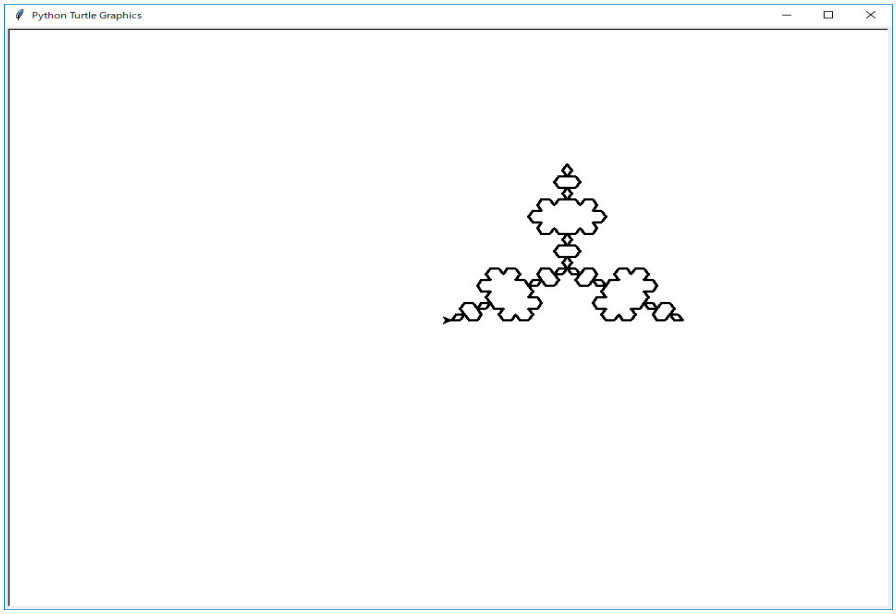
とすれば、



を描きます。

```
from turtle import *
def koch(size, level):
    if level == 0:
        fd(size)
        return
    koch(size/3, level-1)
    lt(60)
    koch(size/3, level-1)
    rt(120)
    koch(size/3, level-1)
    lt(60)
    koch(size/3, level-1)
pensize(3)
for _ in range(3):
    koch(250, 3)
    lt(120)
```

とすれば、



を描きます。

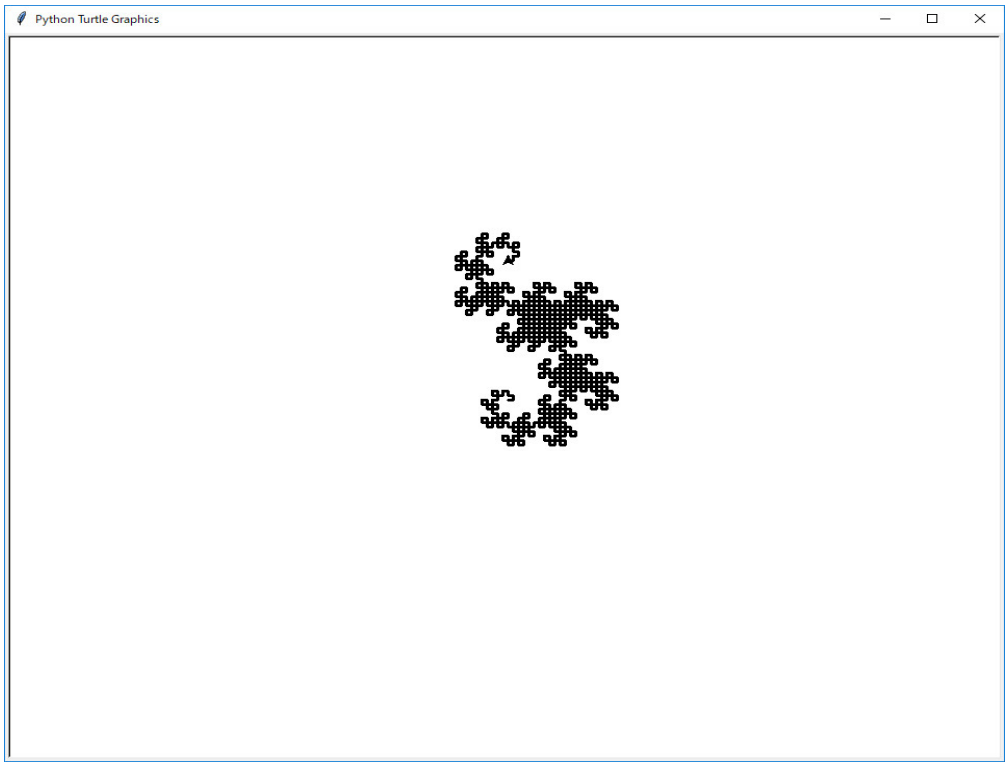
```
def rdragon(size, level):
    if level == 0:
        fd(size)
        return
    ldragon(size, level-1)
    rt(90)
    rdragon(size, level-1)

def ldragon(size, level):
    if level == 0:
        fd(size)
        return
    ldragon(size, level-1)
    lt(90)
    rdragon(size, level-1)
```

と定義して、

```
from turtle import *
def rdragon(size, level):
    if level == 0:
        fd(size)
        return
    ldragon(size, level-1)
    rt(90)
    rdragon(size, level-1)
```

```
def ldragon(size, level):
    if level == 0:
        fd(size)
        return
    ldragon(size, level-1)
    lt(90)
    rdragon(size, level-1)
pensize(3)
ldragon(5, 10)
```

とすれば、



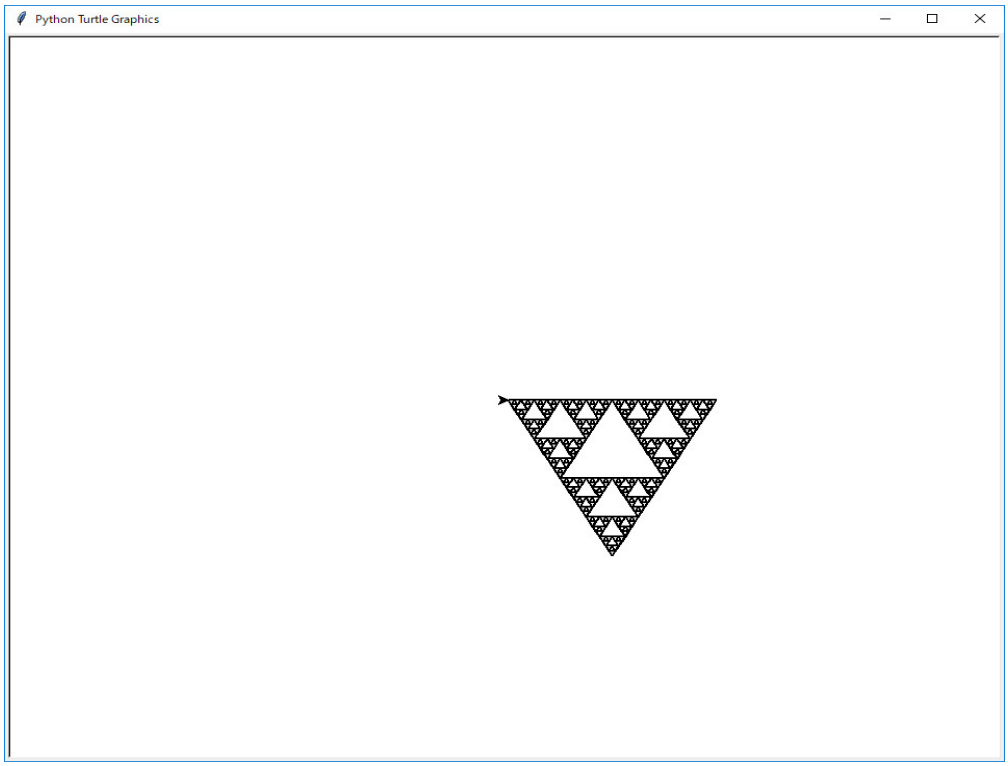
を描きます。ドラゴン曲線と言います。

```
def nested_triangle(size):
    if size < 5:
        return
    for _ in range(3):
        nested_triangle(size/2)
        fd(size)
        rt(120)
```

と定義して、

```
from turtle import *
def nested_triangle(size):
    if size < 5:
        return
    for _ in range(3):
        nested_triangle(size/2)
        fd(size)
        rt(120)
pensize(2)
nested_triangle(200)
```

とすれば、



を描きます。

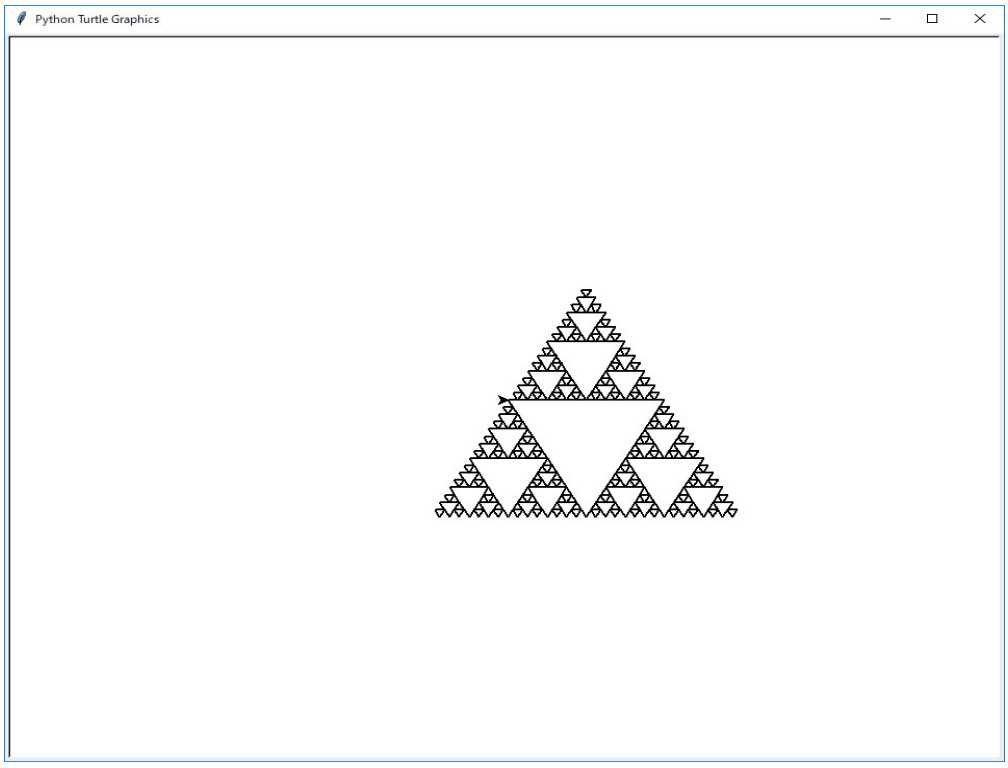
```
def insert(size):
    lt(120)
    outward_triangle(size/2)
    rt(120)

def outward_triangle(size):
    if size < 5:
        return
    for _ in range(3):
```

```
        fd(size/2)
        insert(size)
        fd(size/2)
        rt(120)
```

と定義して、

```
from turtle import *
def insert(size):
    lt(120)
    outward_triangle(size/2)
    rt(120)

def outward_triangle(size):
    if size < 5:
        return
    for _ in range(3):
        fd(size/2)
        insert(size)
        fd(size/2)
        rt(120)
pensize(2)
outward_triangle(150)
```

とすれば、



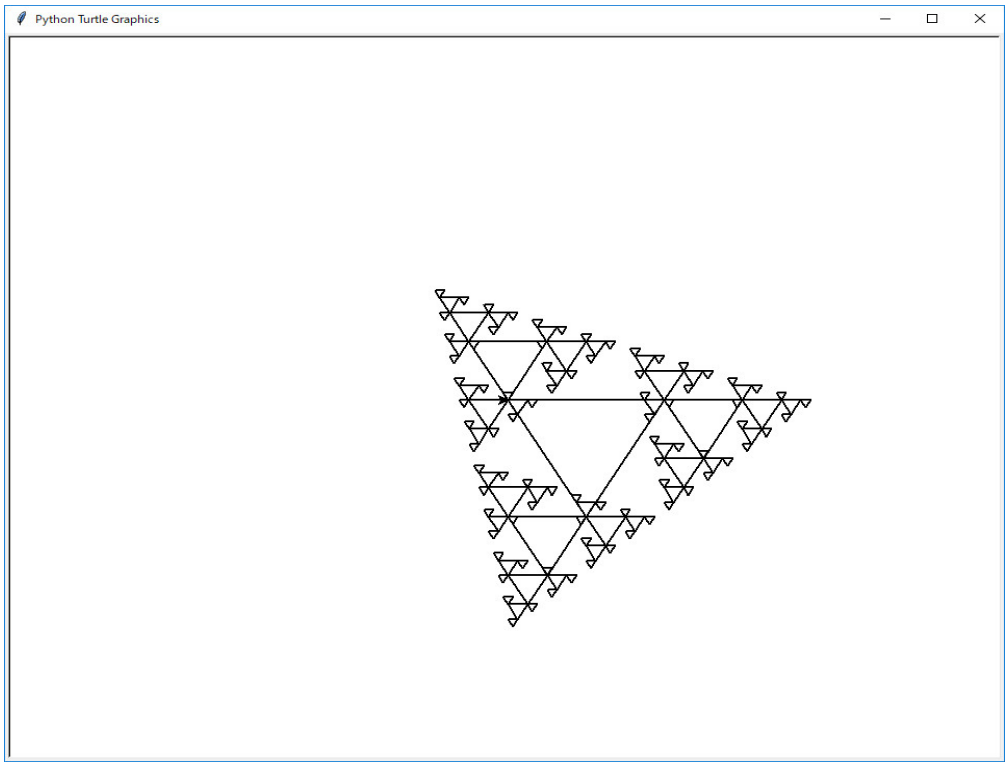
を描きます。

```
def corner_triangle(size):
    if size < 5:
        return
    for _ in range(3):
        fd(size)
        corner_triangle(size/2)
        rt(120)
```

と定義して、

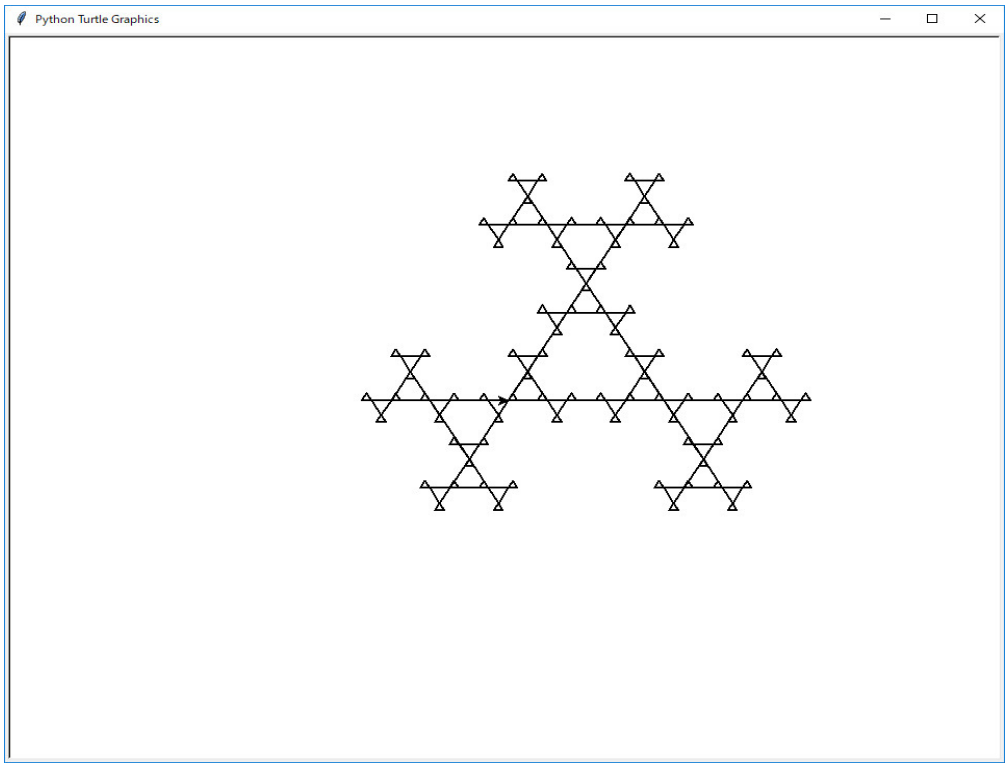
```
from turtle import *
def corner_triangle(size):
    if size < 5:
        return
    for _ in range(3):
        fd(size)
        corner_triangle(size/2)
        rt(120)
pensize(2)
corner_triangle(150)
```

とすれば、



を描きます。

問題：



のような図を描くプログラムを作りなさい。

解答例：

```
from turtle import *
def corner_triangle(size):
    if size < 5:
        return
    for _ in range(3):
        lt(60)
        fd(size)
        corner_triangle(size/2)
        rt(60)
        rt(120)
pensize(2)
corner_triangle(150)

def lhilbert(size, level):
    if level == 0:
        return
    lt(90)
    rhilbert(size, level-1)
    fd(size)
```

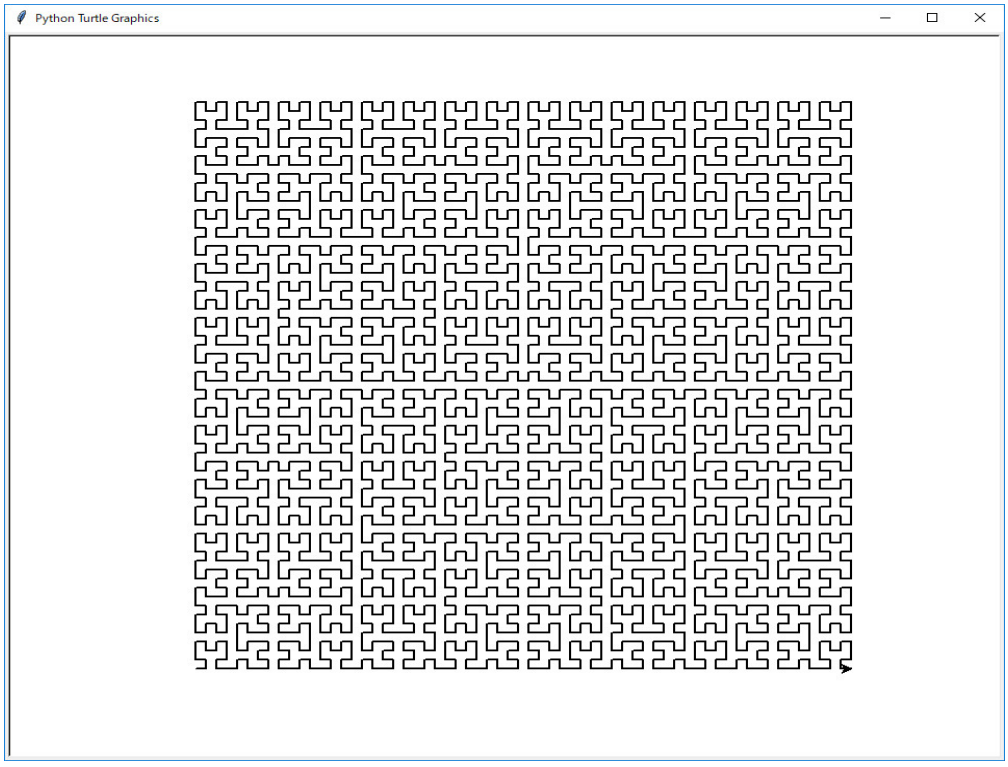
```
    rt(90)
    lhilbert(size, level-1)
    fd(size)
    lhilbert(size, level-1)
    rt(90)
    fd(size)
    rhilbert(size, level-1)
    lt(90)

def rhilbert(size, level):
    if level == 0:
        return
    rt(90)
    lhilbert(size, level-1)
    fd(size)
    lt(90)
    rhilbert(size, level-1)
    fd(size)
    rhilbert(size, level-1)
    lt(90)
    fd(size)
    lhilbert(size, level-1)
    rt(90)
```

と定義して、

```
from turtle import *
def lhilbert(size, level):
    if level == 0:
        return
    lt(90)
    rhilbert(size, level-1)
    fd(size)
    rt(90)
    lhilbert(size, level-1)
    fd(size)
    lhilbert(size, level-1)
    rt(90)
    fd(size)
    rhilbert(size, level-1)
    lt(90)

def rhilbert(size, level):
    if level == 0:
```

```
        return
    rt(90)
    lhilbert(size, level-1)
    fd(size)
    lt(90)
    rhilbert(size, level-1)
    fd(size)
    rhilbert(size, level-1)
    lt(90)
    fd(size)
    lhilbert(size, level-1)
    rt(90)
pensize(2)
pu()
setpos(-300,-300)
pd()
lhilbert(10, 6)
```

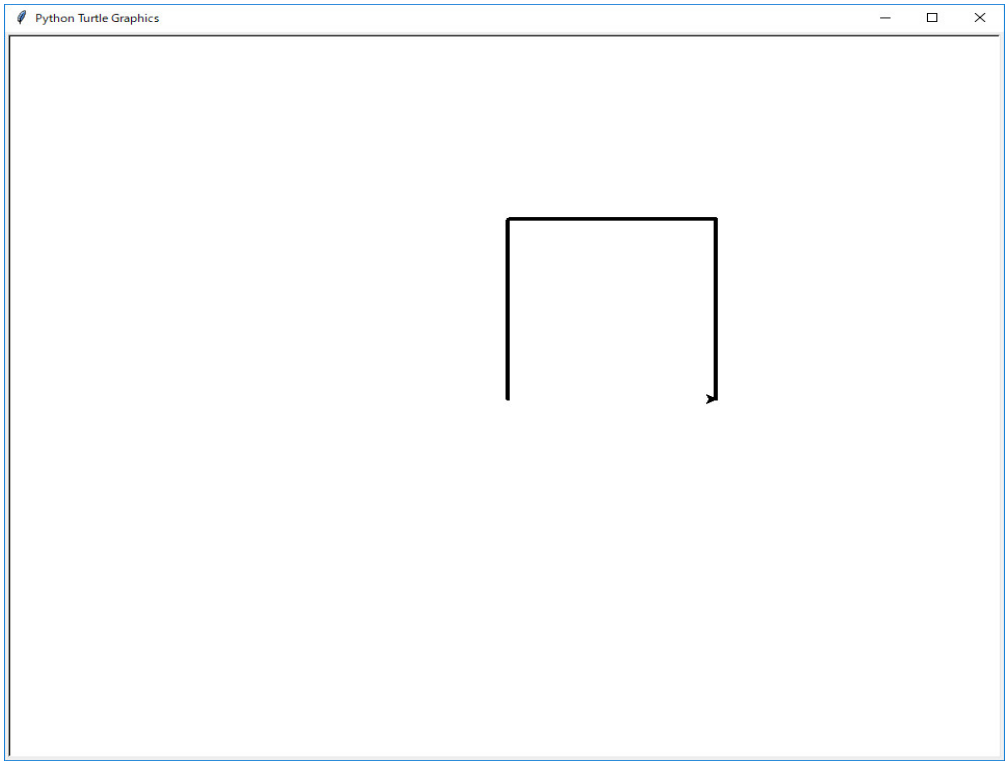
とすれば、



を描きます。ヒルベルト曲線と言います。複雑ですが

```
lhilbert(200 1)
```

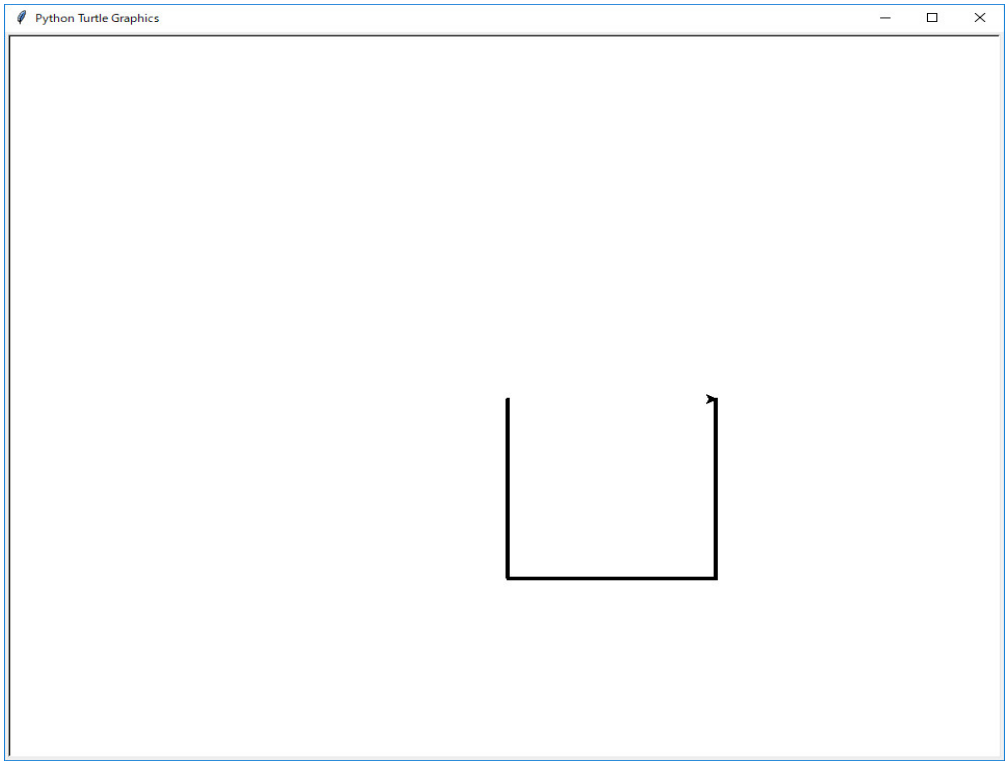
は



を描きます。

`rhilbert(200 1)`

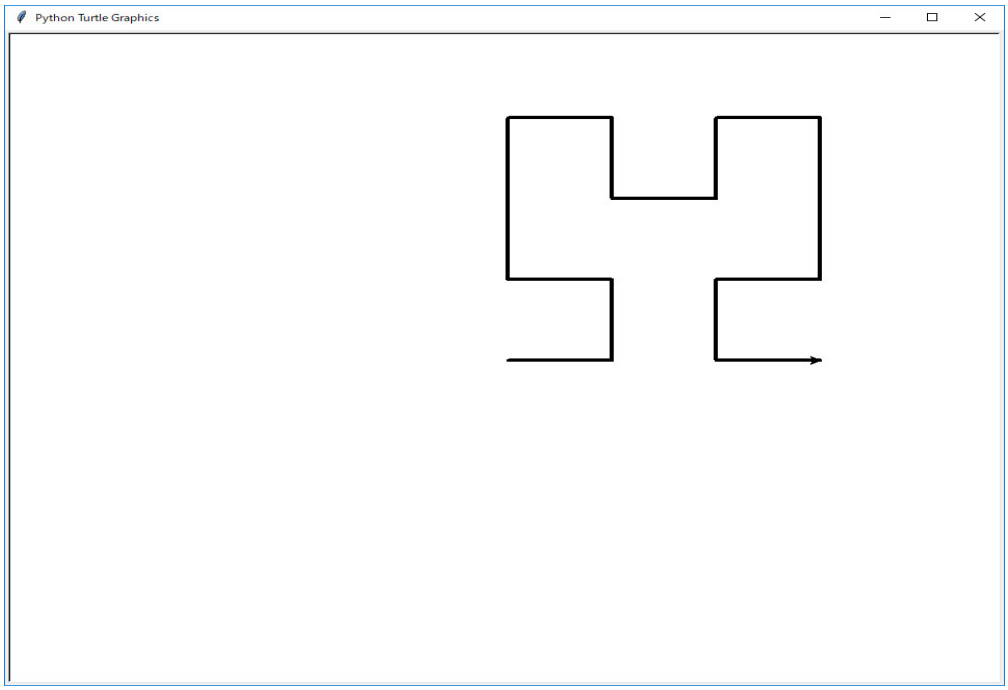
は



を描きます。

```
lhilbert(100 2)
```

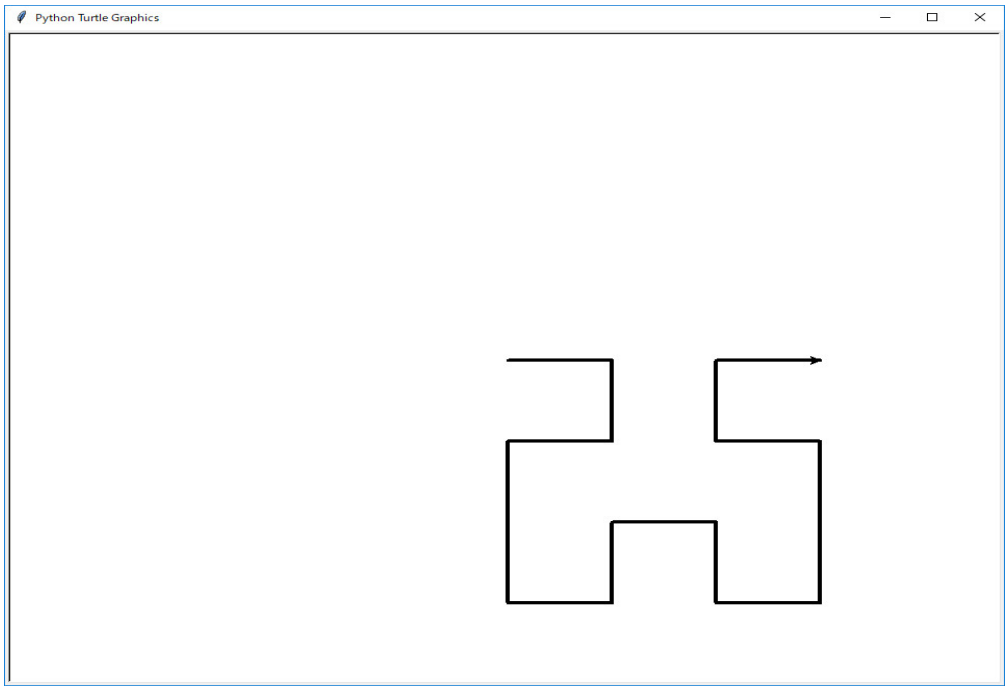
は



を描きます。

```
rhilbert(100 2)
```

は



を描きます。このように小さいレベルの図を描いてみると、ヒルベルト曲線を描く規則が分かります。

```
def A(size, level):
    if level == 0:
        return
    A(size, level-1)
    rt(45)
    fd(size)
    rt(45)
    B(size, level-1)
    lt(90)
    fd(2*size)
    lt(90)
    D(size, level-1)
    rt(45)
    fd(size)
    rt(45)
    A(size, level-1)

def B(size, level):
    if level == 0:
        return
    B(size, level-1)
    rt(45)
    fd(size)
    rt(45)
    C(size, level-1)
    lt(90)
    fd(2*size)
    lt(90)
    A(size, level-1)
    rt(45)
    fd(size)
    rt(45)
    B(size, level-1)

def C(size, level):
    if level == 0:
        return
    C(size, level-1)
    rt(45)
    fd(size)
```

```
    rt(45)
    D(size, level-1)
    lt(90)
    fd(2*size)
    lt(90)
    B(size, level-1)
    rt(45)
    fd(size)
    rt(45)
    C(size, level-1)

def D(size, level):
    if level == 0:
        return
    D(size, level-1)
    rt(45)
    fd(size)
    rt(45)
    A(size, level-1)
    lt(90)
    fd(2*size)
    lt(90)
    C(size, level-1)
    rt(45)
    fd(size)
    rt(45)
    D(size, level-1)

def sierpinski(size, level):
    A(size, level)
    rt(45)
    fd(size)
    rt(45)
    B(size, level)
    rt(45)
    fd(size)
    rt(45)
    C(size, level)
    rt(45)
    fd(size)
    rt(45)
    D(size, level)
    rt(45)
```

```
    fd(size)
    rt(45)
```

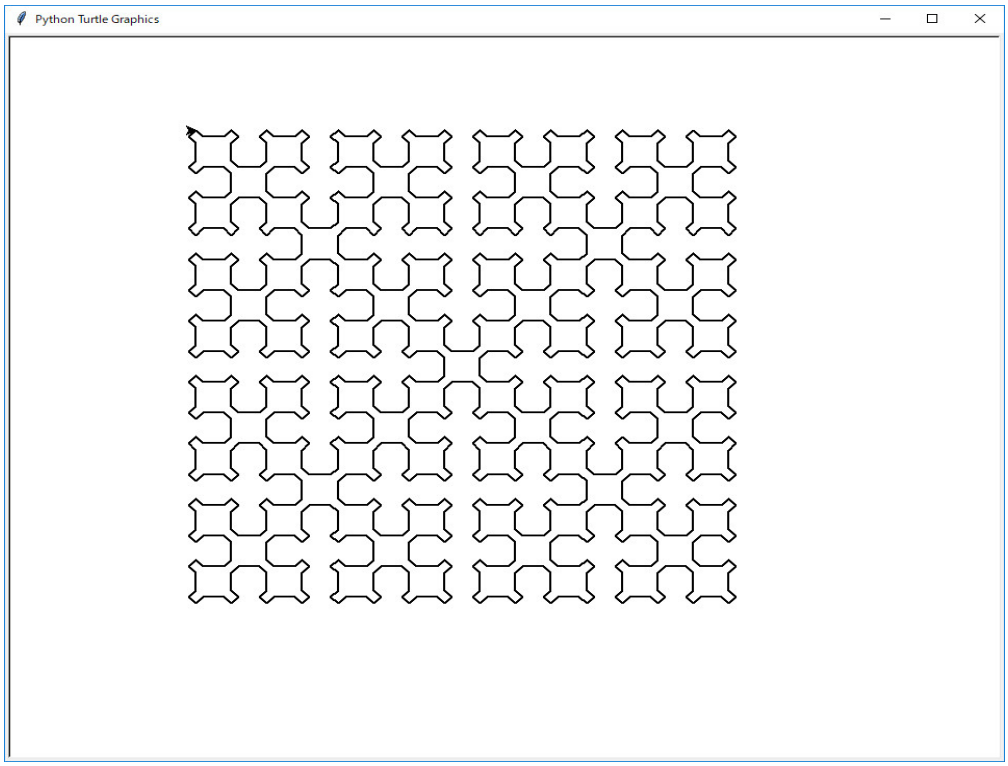
と定義して、

```
from turtle import *
def A(size, level):
    if level == 0:
        return
    A(size, level-1)
    rt(45)
    fd(size)
    rt(45)
    B(size, level-1)
    lt(90)
    fd(2*size)
    lt(90)
    D(size, level-1)
    rt(45)
    fd(size)
    rt(45)
    A(size, level-1)

def B(size, level):
    if level == 0:
        return
    B(size, level-1)
    rt(45)
    fd(size)
    rt(45)
    C(size, level-1)
    lt(90)
    fd(2*size)
    lt(90)
    A(size, level-1)
    rt(45)
    fd(size)
    rt(45)
    B(size, level-1)

def C(size, level):
    if level == 0:
        return
    C(size, level-1)
```

```
    rt(45)
    fd(size)
    rt(45)
    D(size, level-1)
    lt(90)
    fd(2*size)
    lt(90)
    B(size, level-1)
    rt(45)
    fd(size)
    rt(45)
    C(size, level-1)

def D(size, level):
    if level == 0:
        return
    D(size, level-1)
    rt(45)
    fd(size)
    rt(45)
    A(size, level-1)
    lt(90)
    fd(2*size)
    lt(90)
    C(size, level-1)
    rt(45)
    fd(size)
    rt(45)
    D(size, level-1)

def sierpinski(size, level):
    A(size, level)
    rt(45)
    fd(size)
    rt(45)
    B(size, level)
    rt(45)
    fd(size)
    rt(45)
    C(size, level)
    rt(45)
    fd(size)
    rt(45)
```

```
    D(size, level)
    rt(45)
    fd(size)
    rt(45)
pensize(2)
pu()
setpos(-300,300)
pd()
sierpinski(10, 4)
```

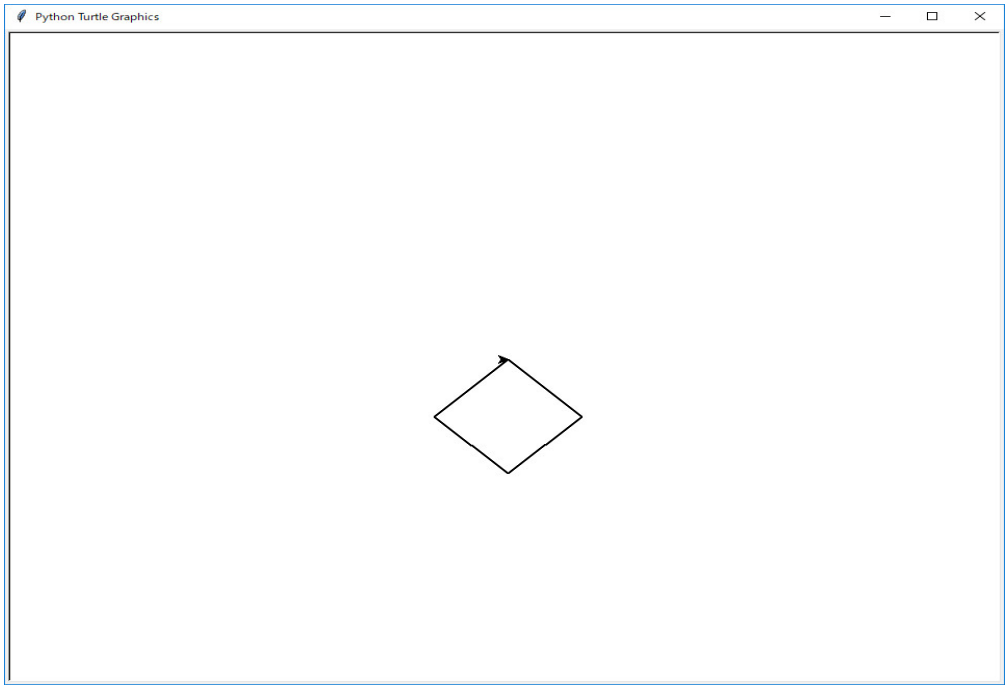
とすれば、



を描きます。シェルピンスキー曲線と言います。シェルピンスキー曲線を描く規則が分かりますか?

```
sierpinski(100, 0)
```

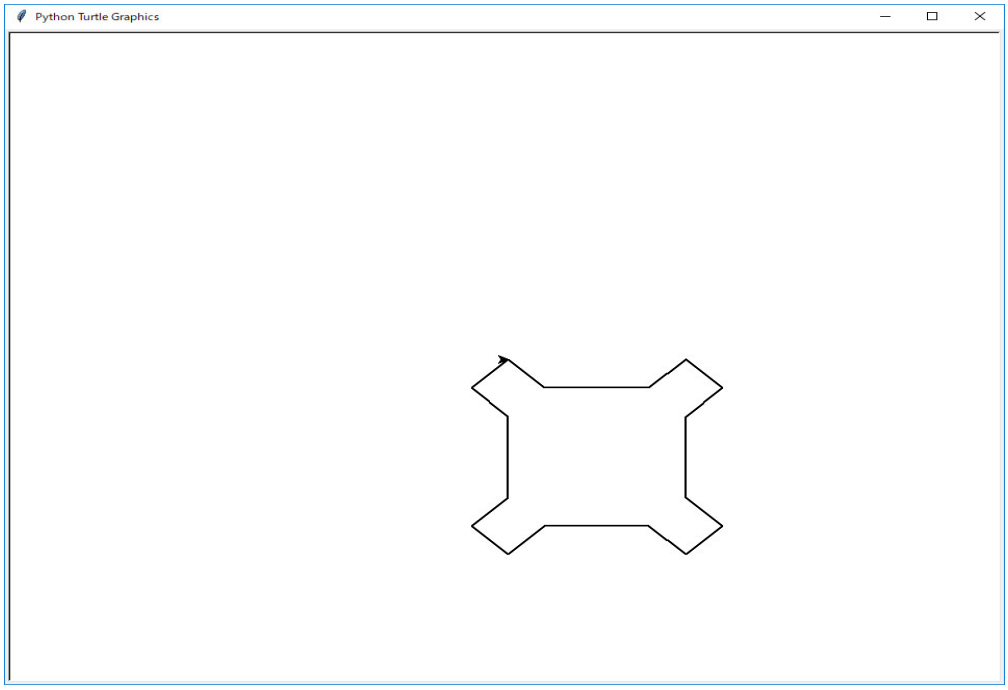
は



を描きます。

```
sierpinski(100, 1)
```

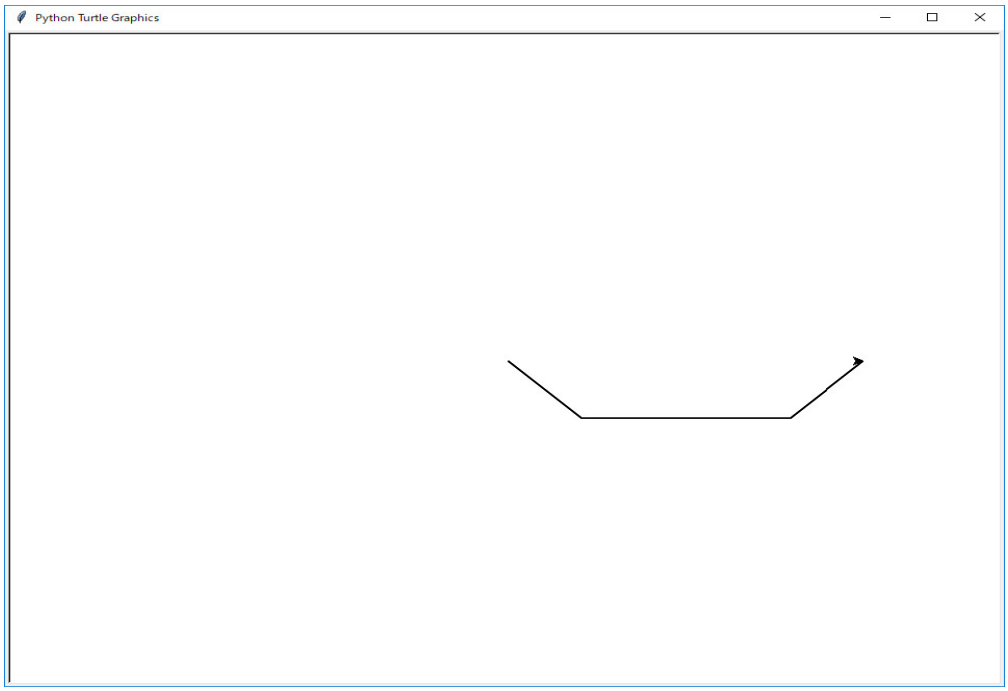
は



を描きます。

```
A(100, 1)
```

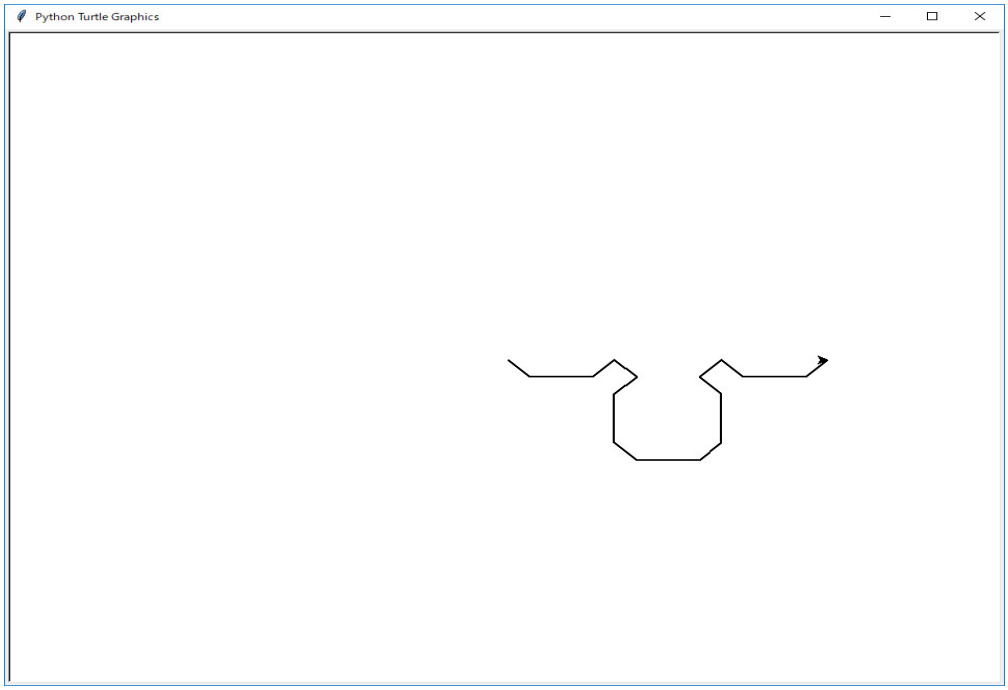
は



を描きます。

```
A(30, 2)
```

は



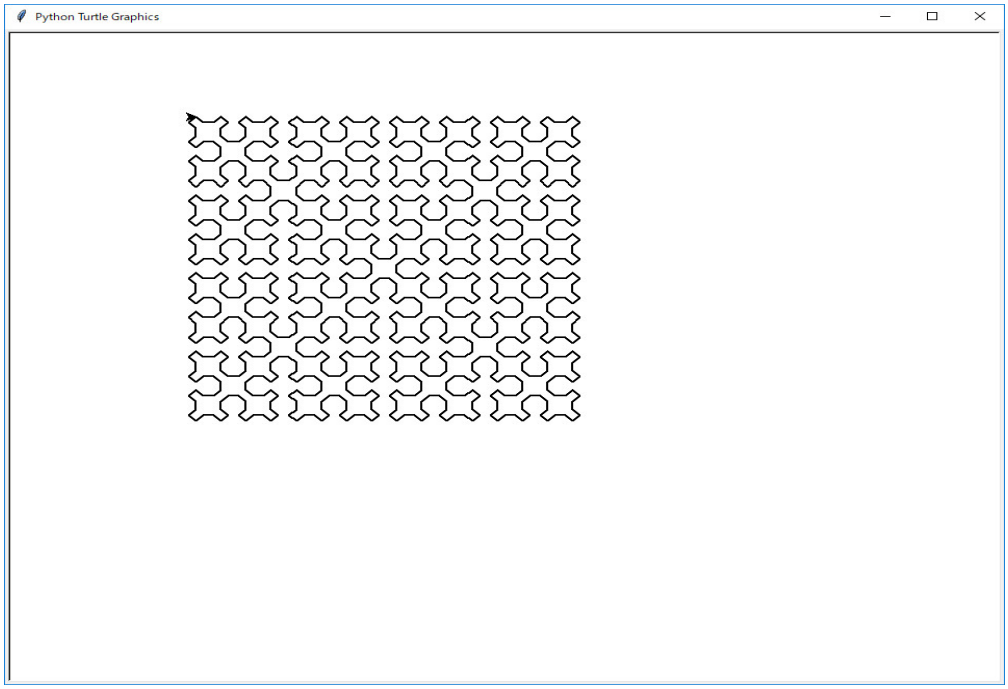
を描きます。このようにして見ていくと再帰の規則が見えてきます。

```
from turtle import *
def A(size, level):
    if level == 0:
        return
    A(size, level-1)
    rt(45)
    fd(size)
    rt(45)
    B(size, level-1)
    lt(90)
    fd(size)
    lt(90)
    D(size, level-1)
    rt(45)
    fd(size)
    rt(45)
    A(size, level-1)

def B(size, level):
    if level == 0:
        return
    B(size, level-1)
    rt(45)
    fd(size)
    rt(45)
    C(size, level-1)
    lt(90)
    fd(size)
    lt(90)
    A(size, level-1)
    rt(45)
    fd(size)
    rt(45)
    B(size, level-1)

def C(size, level):
    if level == 0:
        return
    C(size, level-1)
    rt(45)
    fd(size)
    rt(45)
    D(size, level-1)
```

```
    lt(90)
    fd(size)
    lt(90)
    B(size, level-1)
    rt(45)
    fd(size)
    rt(45)
    C(size, level-1)

def D(size, level):
    if level == 0:
        return
    D(size, level-1)
    rt(45)
    fd(size)
    rt(45)
    A(size, level-1)
    lt(90)
    fd(size)
    lt(90)
    C(size, level-1)
    rt(45)
    fd(size)
    rt(45)
    D(size, level-1)

def sierpinski(size, level):
    A(size, level)
    rt(45)
    fd(size)
    rt(45)
    B(size, level)
    rt(45)
    fd(size)
    rt(45)
    C(size, level)
    rt(45)
    fd(size)
    rt(45)
    D(size, level)
    rt(45)
    fd(size)
    rt(45)
```

```
pensize(2)
pu()
setpos(-300,300)
pd()
sierpinski(10, 4)
```

と修正すると



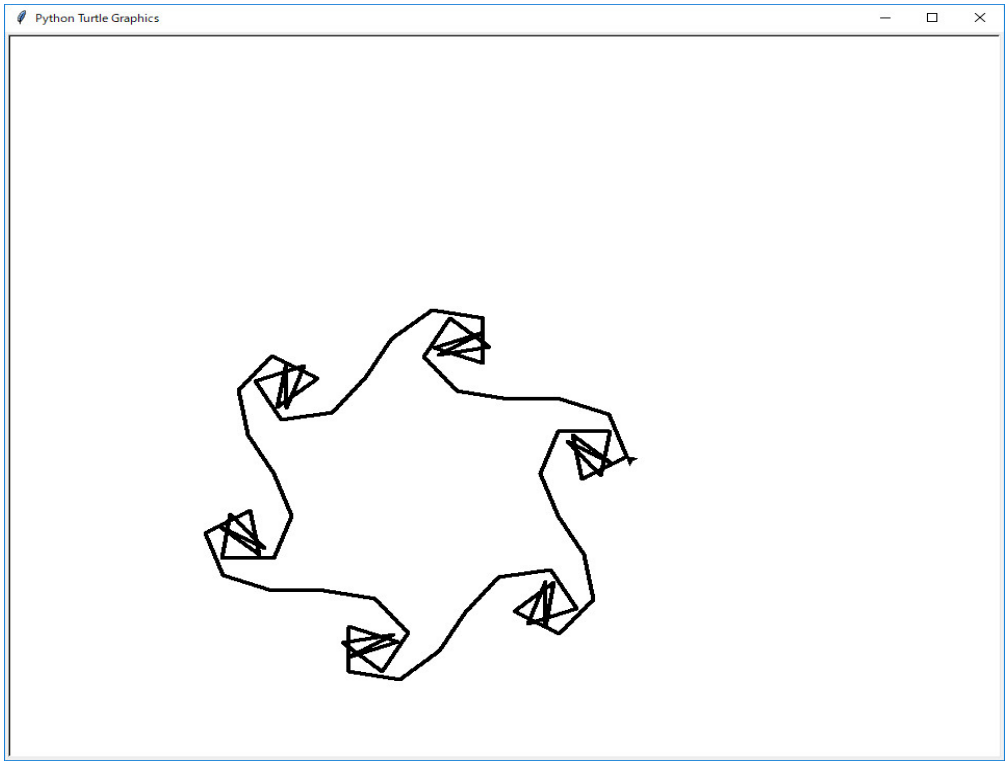
を描きます。
　最後に

```
def poly(size, angle, increment):
    forward(size)
    right(angle)
    poly(size, angle+increment, increment)
```

と定義します。この定義は欠陥があって、いつまでたっても停止しません。Python.exe を実行した画面をアクティブにして、CNTL キーを押しながら、C のキーを押して停止してください。

```
from turtle import *
def poly(size, angle, increment):
    forward(size)
    right(angle)
    poly(size, angle+increment, increment)
poly(20, 20, 30)
```

を実行すると



こんな絵を描きます。angle と increment の組み合わせで、色々なパターンの図を描きます。古い本ですが、Harold Abelson and Andrea diSessa : TURTLE GEOMETRY (The Computer as a Medium for Exploring Mathmatics), The MIT Press という本の中にあった例題です。TURTLE GEOMETRY が一番数学的なタートルグラフィックスの参考書です。TURTLE GEOMETRY の本は私が購入していましたから、図書館にあるはずです。タートルグラフィックスの例は私の Kite というソフトの解説の中にもあります。kite2 は球面上のタートルグラフィックスの、kite3 が非ユークリッド空間のタートルグラフィックスのためのソフトです。

問題：

```
def poly(size, angle, increment):
    forward(size)
    right(angle)
    poly(size, angle+increment, increment)
```

のプラグラムをきちんと停止するように修正し、それが停止する理由を説明しなさい。これが出来たら無条件で優です。

問題：

小学生が喜ぶような絵を描くプログラムを作れ。

先輩が作った絵の例は私の Kite というソフトの解説の中にあります。

これで Python の turtle graphics は終わりです。