オブジェクト指向プログラミング

第６回

箕原辰夫

オブジェクトの生成

- 書式：クラス名（生成時のパラメータ)

  Tk( ) … tkinterでウィンドウを作る

  Canvas(*window*) … 指定したウィンドウ上にキャンバスを作る

  Font(family="Helvetica")…フォントのオブジェクトを作る

  C++ / C# /Java/JavaScript: **new** クラス名（生成時のパラメータ)

オブジェクトを参照する変数

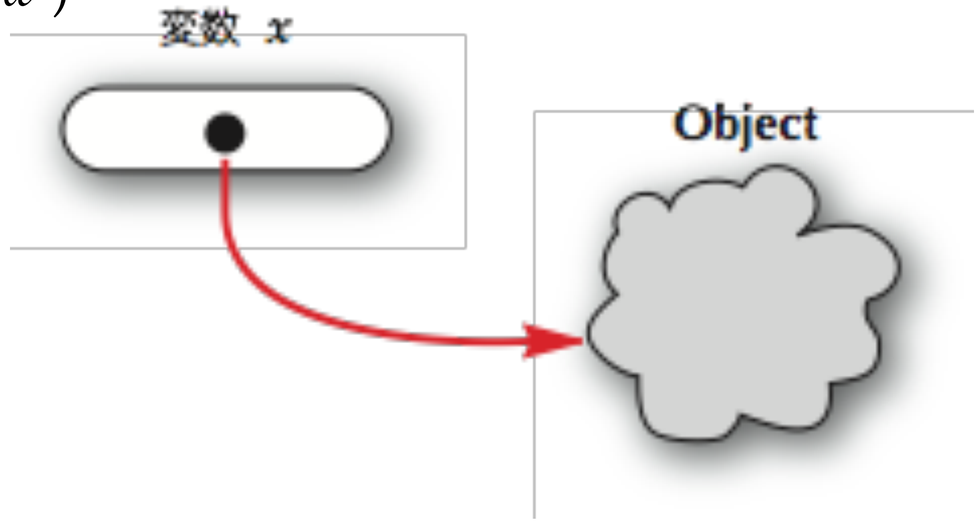
- 代入
  - 変数名 = クラス名（パラメータ)
  - 例：*window* = Tk( )
  - *r* = range(1, 10)
  - *enums* = enumerate(*nlist*)
  - *canvas* = Canvas(*window*)
  - *font* = Font(family="Helvetica")

オブジェクトの参照

変数= クラス名（パラメータ）

例：

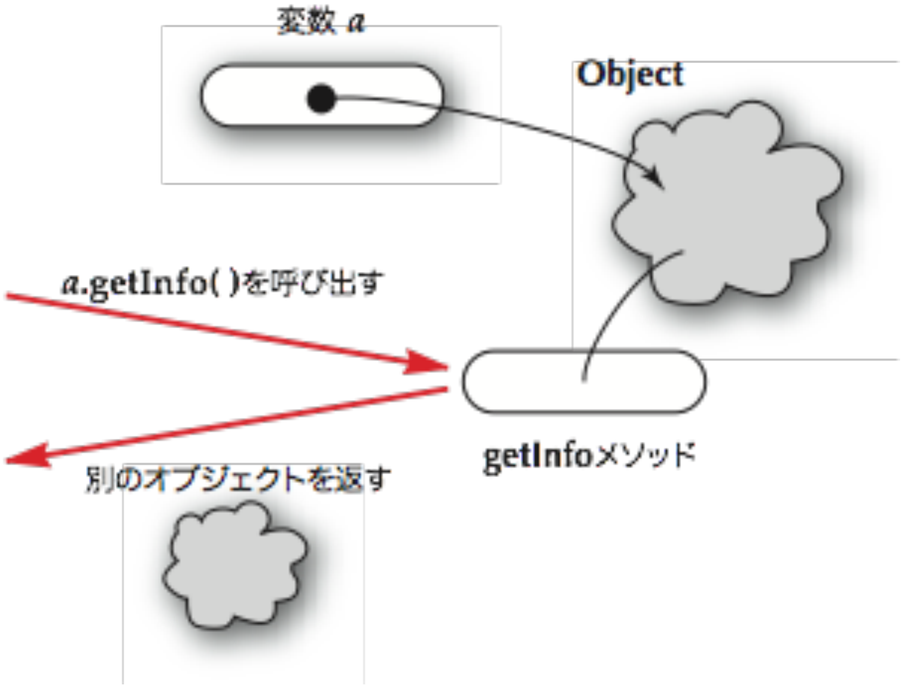
window = Tk( )
x = Canvas(*window*)



オブジェクト変数の利用法

- メソッド（オブジェクトが保持する関数）を呼び出す
  - 変数.メソッド名（実パラメータ）
  - 例： *x*.create_line(50, 50, 30, 30)

    another = *a*.getInfo( )



1回限りのオブジェクト

- パラメータの引数に使っても良い
  - c.create_text(x, y, font=Font(family="Times", size= 35))
- 変数に代入したのと同じ効果がある
  - timesfont=Font(family="Times", size= 35)
  - c.create_text(x, y, font=timesfont)
- ドット記法も使える
  - f = Font(family="Times", size= 35)
  - as = f.metrics(ascent)
  - as = (Font(family="Times", size=35)).metrics(ascent)

オブジェクトの自動消去と手動消去

- ガーベージコレクタが自動的にやってくれる

- 明示的にオブジェクトを削除する場合は、del文を使う

  del *window*

tkを使って描画する

- **from** tkinter **import** * # 標準のGUIライブラリ

- トップのウィンドウを作って配置する

  window = Tk()

  window.geometry("500x500+100+100") # 小文字のx（エックス）

  ‣# 幅と高さ、左上の角のデスクトップ上の位置

  window.attributes("-topmost", True)

  ‣# 表示を前面に

- このあと、GUIのコンポーネント（ボタンなど）を動作させるときは、mainloop関数を呼び出す。

  mainloop()

キャンバスの作成と配置

- 描画用のCanvasをウィンドウ上に作成する
- widthはキャンバスの幅、heightはキャンバスの高さ
  - c = Canvas(window, width= 500, height=500)
  - c.pack()で描画を確定させる

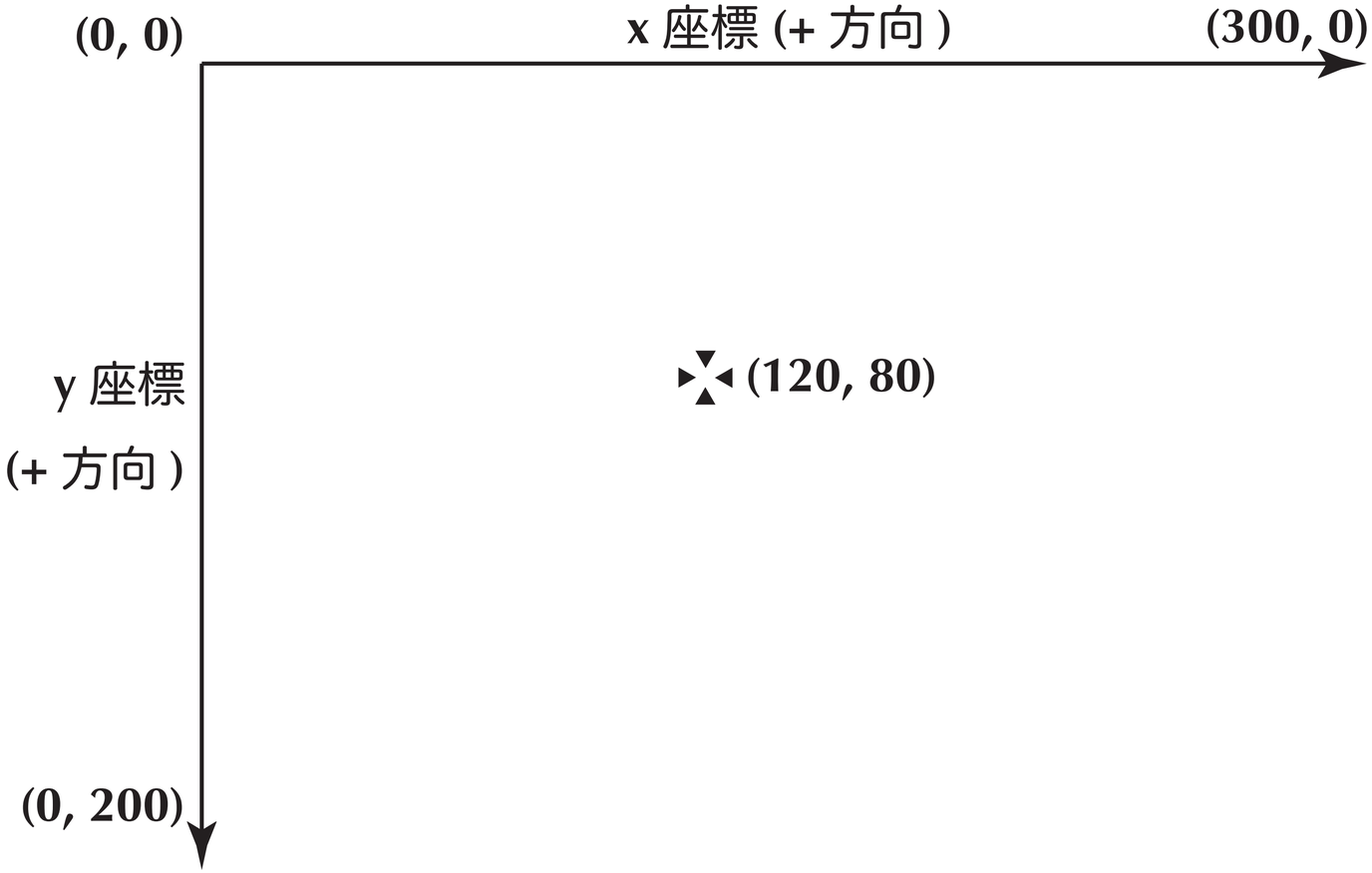
- 描画関数は、このキャンバスのオブジェクトへの関数呼出しとして記述される。
- 例：

```
c.create_line( 100, 100, 200, 200 )
```

ウィンドウとキャンバスのその他の関数

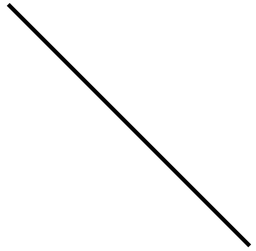
- ウィンドウ
  - title(ウィンドウのタイトル)
  - geometry("幅x高さ") # 半角小文字のx（エックス）
- ウィンドウの幅と高さを求める
  - winfo_width()
  - winfo_height()
  - update() してからでないと求まらない
- キャンバス
  - place(x=左側の開始位置, y=上側の開始位置)
  - 作成時のbackgroundオプション

ウィンドウ上の座標系

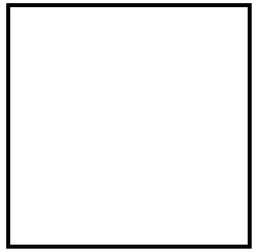


tkinterの描画メソッド（１）

- Canvasオブジェクトに対して、描画関数を呼び出す
  - create_line(x1, y1, x2, y2)
    - 始点から終点に向けて、線を描く
    - 始点と終点を取り替えても可能



  - create_rectangle(x1, y1, x2, y2)
    - 1番目と2番目は左上の座標
    - ３番目と４番目は右下の座標

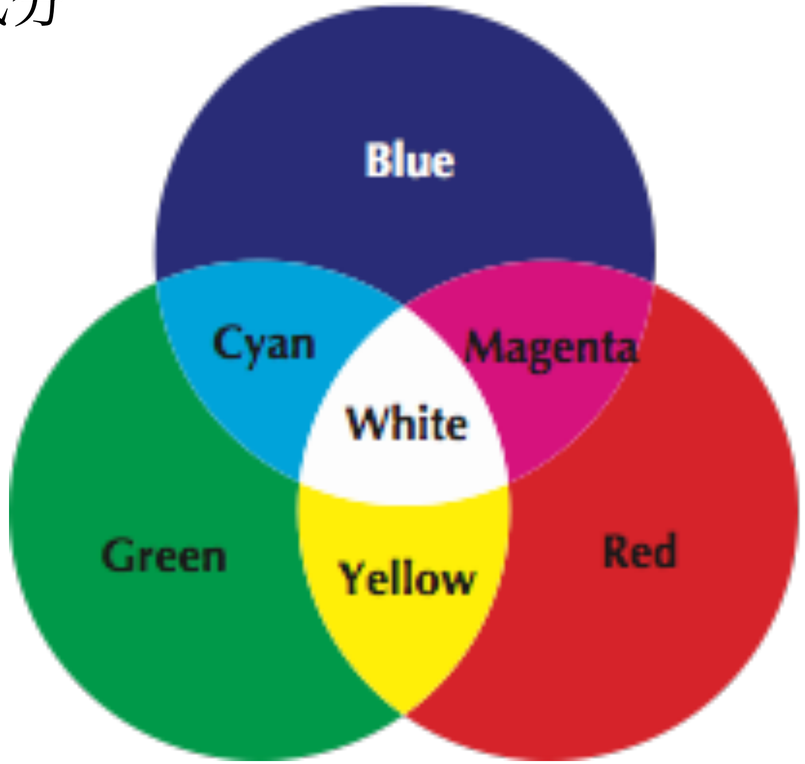


オプションの指定

- fill = 色
  - 内部を塗りつぶす色を文字列で指定する
  - 指定できる色の一覧は、以下のサイトを参照
    http://www.tcl.tk/man/tcl8.4/TkCmd/colors.htm
  - 16進数3桁あるいは6桁でも指定できる#から始める
    例： #f3a　←rgb1桁ずつ　#33ffcc　　←rgb2桁ずつ
- outline = 色
  - 枠の色を文字列で指定する
- width = 幅
  - 枠の幅（デフォルトは 1.0）を数値で指定する

RGBカラーモデル

- RGBカラーモデルで
  - #赤成分 緑成分 青成分

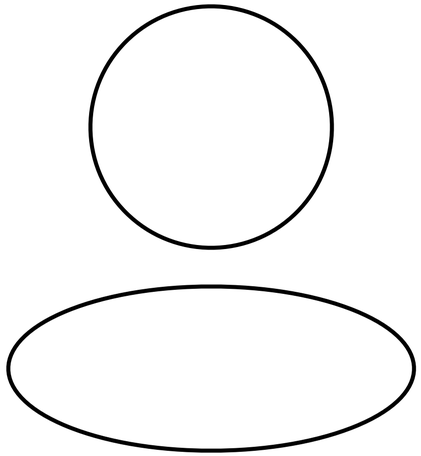


16進数によるカラー指定

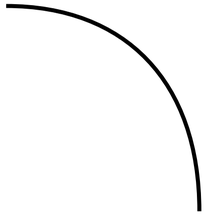
- 各色の成分指定による生成（加法混色）
  - 赤 #ff0000　#f00
  - 緑 #00ff00　#0f0
  - 青 #0000ff　#00f
  - シアン #00ffff　#0ff
  - 黄 #ffff00　#ff0
  - マゼンタ #ff00ff　#f0f
  - 黒 #000000　#000
  - 白 #ffffff　#fff

tkinterの描画メソッド（２）

- create_oval(x1, y1, x2, y2)
  - 外接する四角形はcreate_rectangleと同じ
  - 幅と高さが同じだと正円
  - 幅と高さが異なると楕円



- create_arc(x1, y1, x2, y2, start=開始角, extent=角度差)
  - 四角形に内接する円を描くのはcreate_ovalと同じ
  - 開始角度と角度差を360°で指定
  - style=スタイルで、ARC, CHORD
    PIESLICEで指定できる



tkinterの描画メソッド（３）

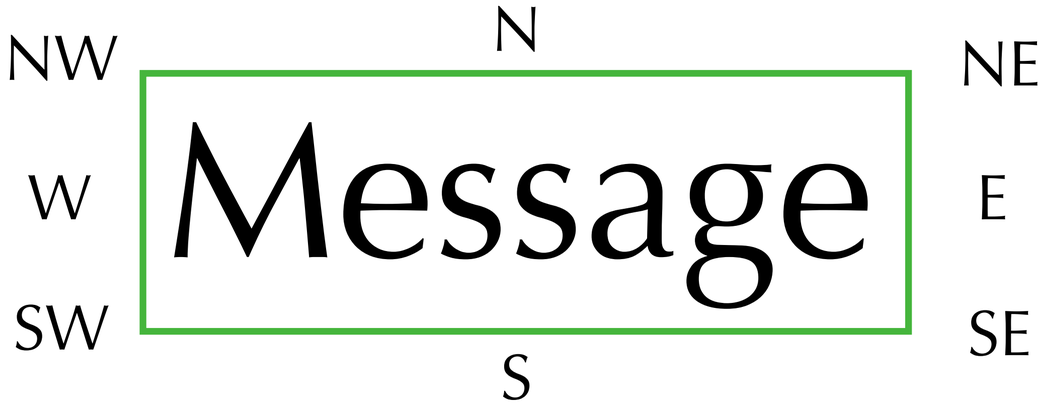
- create_line(x1, y1, x2, y2, x3, y3, …)
  - 1本の線だけでなく、折れ線で連続した線も描画することができる
  - smooth=Trueオプションで、曲線にすることも可能

- create_polygon(x1, y1, x2, y2, x3, y3, …)
  - create_lineと同じだが、閉包多角形のなので、最低6つのパラメータが必要
  - 6つで、三角形、２つパラメータが増えるごとに四角形、五角形、六角形と増えていく

tkinterの描画メソッド（4）

- create_text(x, y, オプション…)
- オプションは以下のようなものがある
  - text="表示するテキスト"
  - justify=LEFT / CENTER / RIGHTのいずれか
  - anchor=指定したx, yの位置がどこかを示す

    通常はCENTERで、NW, N, NE, W, SW, S, SE, Eを指定することができる
  - fill="テキストのカラー"



後からの移動とtkinterの詳細

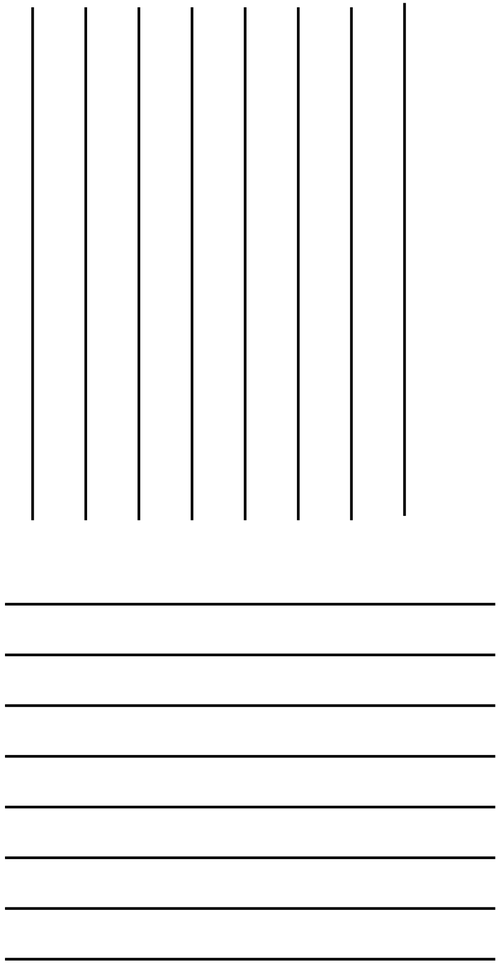
- 描画する前に後から移動することができる
- 描画メソッドがid番号を返してくるので、その番号とmoveメソッドを使って、x, y方向に移動できる
- 例：
  - id = canvas.create_line(100, 100, 200, 200)
  - canvas.move(id, 50, 50)

- tkinterの詳細
  - https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/index.html

    を参照

グラフィックスと繰返し

- create_lineによる繰返し
  - create_line(x1, y1, x2, y2)
- 縦線（垂直線）は、 x 座標が等しい
- 横線（水平線）は、 y 座標が等しい

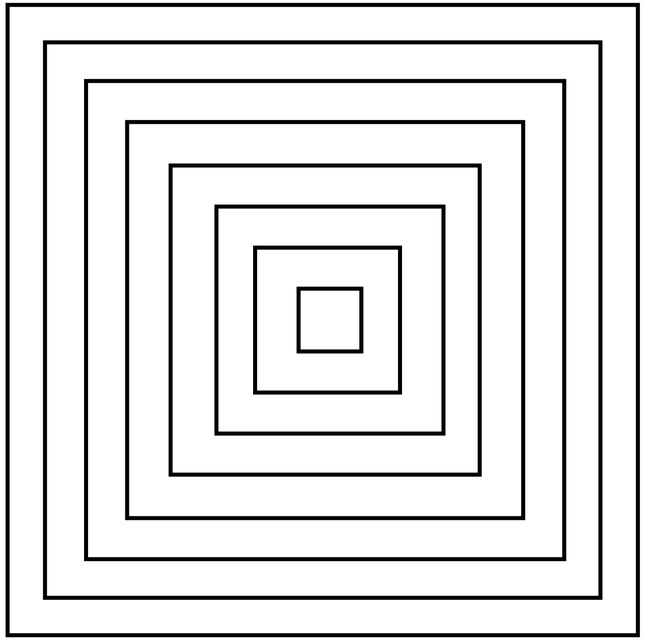


水平線を使った繰返しと描画

- 両端のy座標は、繰返しに併せて変えて行く
  - ループ変数を使う
- x座標を変化させて、動きを示す
  - 両端が同じように変化→平行四辺形、四角形など
  - 違う形で変化→台形、三角形など

その他のグラフィックス・メソッドを使った繰返し

- create_rectangleで、ピラミッドを上から見たような入れ子四角形



カラーテーブル

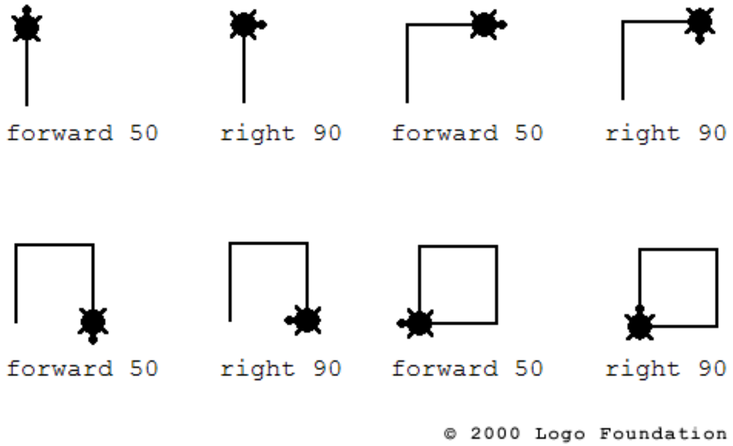
- 赤と青を変えて行きながら描画

補数・逆数を使った繰返し

- 補数
  - 定数からループ変数を引く
  - 100 - 10 * n
- 逆数
  - 定数をループ変数で割る
  - 100 / n
- 補数を使って、階段状の格子図形を作る

タートルグラフィックス

- MIT LabのLOGO言語で開発された描画



turtleライブラリ

- Python2からのものに、オブジェクト指向プログラミングとして、拡張されたPython3版がある。
- ライブラリの採り入れ方

  from turtle **import** *

- Python2版の使い方
  - 以下において、nはピクセル数・angleは角度（360度表記）・x, yはそれぞれウィンドウ上のx, y座標を実パラメータとして与える
    - 前後：　forward(n) | fd(n) | backward(n) | bk(n) | back(n)
    - 回転：　right(angle) | rt(angle) | left(angle) | lt(angle)
    - 位置指定（座標）：　goto(x, y) | setpos(x, y) | setposition(x, y) | home()
    - 位置指定（個別）：　setx(x) | sety(y) | setheading(angle) | seth(angle)
    - 位置を返す：pos( ) | position( ) | heading() | xcor() | ycor()
    - 指定位置までの角度・距離：towards(x, y) | distance(x, y)

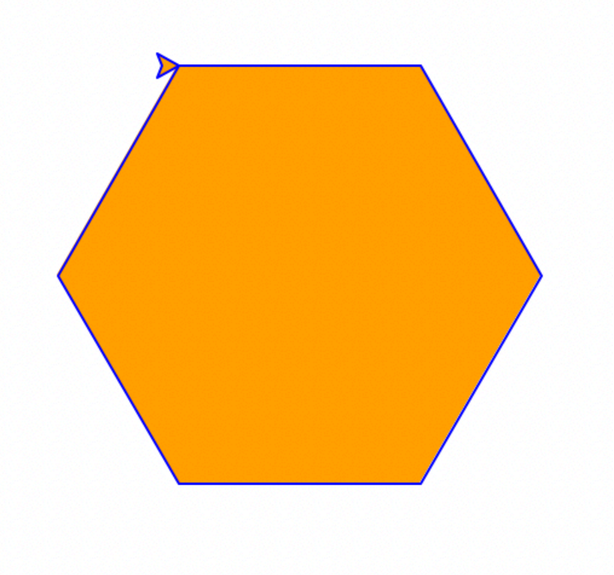
旧来版turtle

- 描画指定・その他の関数
  - 描画：circle(radius) ｜ dot(size [, color])
  - ペンの上げ下げ：penup( ) ｜pendown( ) ｜up() ｜ down() ｜pu() ｜pd()
  - スタンプ：stamp() ｜ clearstamp() ｜ clearstamps(n)
  - 描画色指定： color(line color [, fill color]) … colorは、色名を示す文字列か、rgbの３つ組
  - 塗潰し：begin_fill( ) 〜 end_fill( )
  - その他： undo() ｜ speed(sp) … spは0で最高速、10で高速、1が最低速
  - アニメション省略：tracer(False)…タートルを出さずに高速（描画がきちんとなされていない場合は、ウィンドウのサイズをマウスで変えてみる、ただし最後の方は描画されない）
  - キャンバスサイズの変更： screensize(幅, 高さ)…ただしウィンドウのサイズより大きくしても、ウィンドウのサイズは変わらず、スクロールバーが表示される
  - 画面の消去：clearscreen( )…タートルも原点に戻される
  - ウィンドウの描画ループへの移行： done( )
  - クリックしたら終了する描画ループへの移行：exitonclick( )

旧来版タートル記述例

- 記述例：

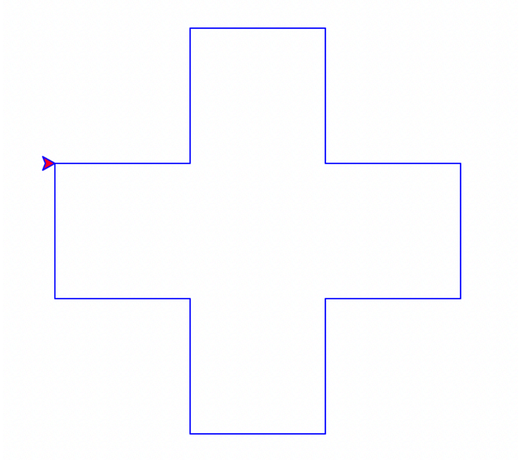
```
from turtle import *
color( "blue", "orange" )
begin_fill( )
for n in range( 6 ):
    forward( 100 )
    right( 60 )
end_fill( )
done( )
```



描画例

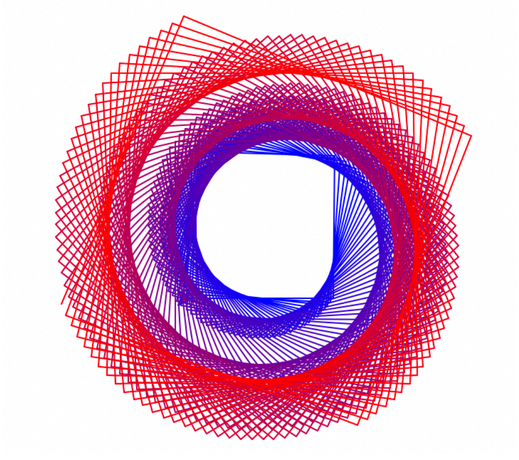
- 十字

```
from turtle import *
color( "blue")
for n in range( 12 ):
    forward( 100 )
    right( 90 if n % 3 != 0 else -90 )
done( )
```



- 色の移り変わり

```
from turtle import *
tracer( False )
for m in range( 128 ):
    color( f"#{m//8:1x}0{15-m//8:1x}" )
    for n in range( 4 ):
        forward( 100+m )
        left( 89.2 )
done( )
```



新版のturtleライブラリ

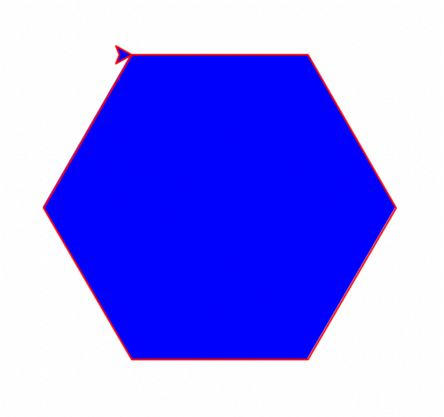
- tkinterとキャンバスとの整合性が図られている
- タートル変数 = RawTurtle(キャンバスオブジェクト)
  - 例：t = RawTurtle(c)
- タートル変数に対して、旧版の関数を用いることができる
  - 例：t.forward(34)
- タートルの状態を知る関数も用意されている
  - position() | pos() … 座標が2つ組みとして返される
  - xcor() | ycor() … 各座標値が返される
  - towards(x, y) … 指定された位置への角度 | heading()…現在のタートルの角度
  - distance(x, y) … 指定された位置までの距離
  - degrees(devider) | radians() … 角度指定の変更
- キャンバスでは、描画終了まで表示されないので、以下のようにして、TurtleScreenでのアニメーションを禁止して表示を早める
  - ts = t.getscreen( )　# 描画スクリーン
  - ts.tracer(0) # アニメーション禁止
- 詳しくは、ドキュメントを参照
  - https://docs.python.jp/3/library/turtle.html

python3版タートル記述例

- tkinterの方のimportを後にする（tkinter側のmainloop()を利用するため）
- 記述例：

```
from turtle import *
from tkinter import *

win = Tk()
win.geometry( "800x800" )
c = Canvas( win, width=800,
height=800 )
```



```
t = RawTurtle( c )
t.color( "red", "blue" )
t.begin_fill()
for n in range( 6 ):
    t.forward( 100 )
    t.right( 60 )
t.end_fill()
c.pack()
mainloop( )
```

オブジェクトの作り方

- コンストラクタで作る

➡ クラス名(パラメータ)

例：Font(family="Arial")

- クラスメソッドから作る

➡ クラス名.getInstance( )

例：Calendar.getInstance( )

- 他のオブジェクトの関数の戻り値として

➡ オブジェクト名.create_クラス名( )　あるいは

➡ オブジェクト名.getクラス名( )

例：*canvas*.createLine(10, 10, 100, 100)

フォントオブジェクト

```
import tkinter.font as tkf
```

tkf.Font(オプションのパラメータ)

Font一覧ソフトウェアでフォントを見てみる

- Mac OS X >> FontBook
- Windows >> Font Viewer Plus

Advanced Font

フォントの種類

- SerifとSans Serif

- ProportionalとMonoSpaced

スタイルとサイズ

- family="フォントファミリー名"
- weight="bold"，weight="normal"
- slant="italic"
- size=ポイント数，　1 point = 1/72 inch = 0.35mm

Times Italic 8pt
Times Italic 10pt
Times Italic 12pt
Times Italic 14pt
Times Italic 16pt
Times Italic 18pt
Times Italic 20pt
Times Italic 24pt
Times Italic 30pt
Times Italic 36pt

Fontオブジェクトを使う

- Fontオブジェクトを作る
  - 例：**import** tkinter.font **as** tkf

    helve36 = tkf.Font(family="Helvetica",
    size=36, weight="bold")

- Fontオブジェクトを描画時に設定する
  - 例：c.create_text(x, y, text="Message",

    font=helve36)

使えるFontファミリーの一覧

- tkinter.font.families()
- コンピュータで使えるフォント名の一覧を、文字列のリストとして返してくれる

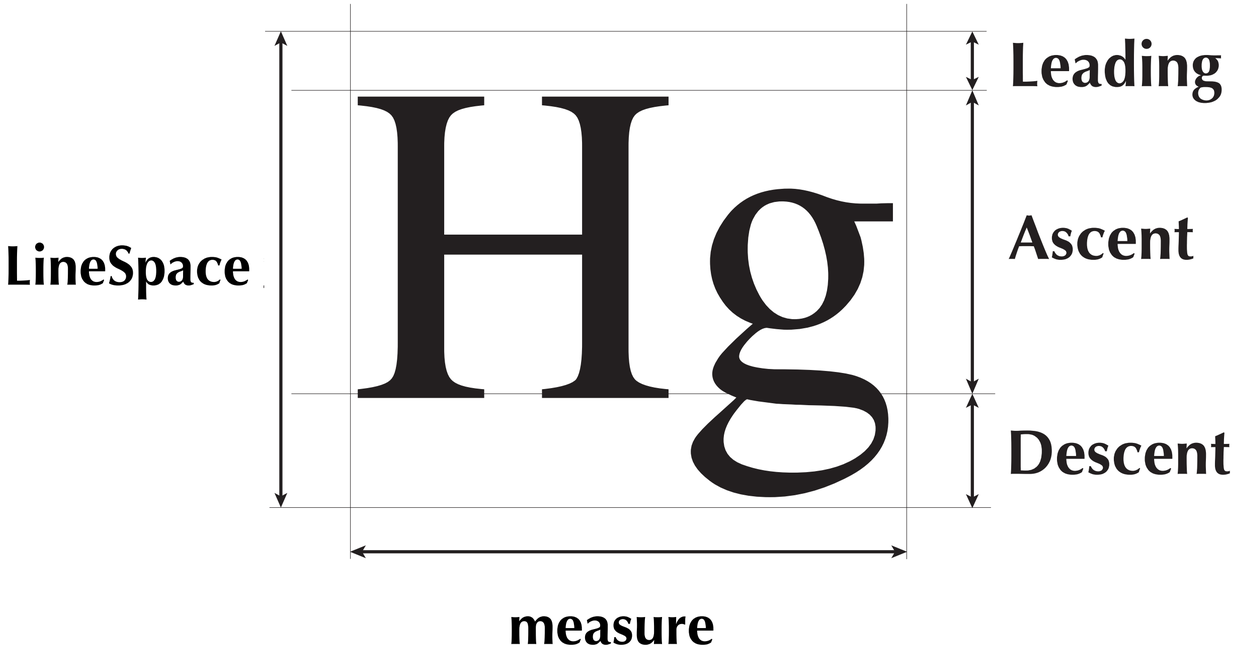
- 使用例：

```
from tkinter import *
import tkinter.font as tkf

w = Tk()
families = tkf.families( )
for fn in families:
    print( fn )
```

フォント情報オブジェクト

- Fontクラスのmetricsメソッドを使って求める
- 幅については、mesureメソッドを使って求める



ピクセルサイズを調べる

- フォントオブジェクト.metrics(オプションパラメータ)
  - オプションパラメータが指定されなければ、すべてが入ったレコードが返される。パラメータは、次のようなものを指定できる
    - "ascent"：ベースラインからの高さ（ピクセル数）
    - "descent"：ベースラインから下のアセンダーまでの高さ（ピクセル数）
    - "fixed"：0…プロポーショナルピッチフォント、
      1…monospaced フォント
    - "linespace"：全体の高さ（leadingも含む）
- フォントオブジェクト.measure(表示文字列)
  - そのフォントで表示する文字列の幅のピクセル数を返す

ライブラリ（パッケージ・モジュール）を使うとき

- いくつかの記述法がある
- **import** [パッケージ名.]モジュール名
  - プログラム中で「モジュール名.名前」で利用可能
  - 例：**import** math; print(math.pi)
- **import** [パッケージ名.]モジュール名 **as** 省略名
  - 「省略名.名前」で利用可能
  - 例：**import** math **as** mt; print(mt.pi)
- **from** [パッケージ名.]モジュール名 **import** クラスあるいは関数あるいは変数（定数）名
  - 「クラス名あるいは関数名あるいは定数名」で利用可能
  - 例：**from** math **import** pi; print(pi)
- **from** [パッケージ名.]モジュール名 **import** *
  - モジュールで定義されているすべての「クラス名あるいは関数名あるいは定数名」で利用可能
  - 例：**from** math **import** * ; print(pi)
- **from** パッケージ名 **import** モジュール名
  - 「モジュール名.名前」で利用可能
  - 例：**from** urllib **import** request;

    request.urlopen("https://www.sfc.keio.ac.jp")

オブジェクトの属性を使う

- いくつかのクラスのオブジェクトでは、オブジェクトが持つ変数を参照することができる
- オブジェクトが持つ変数のことを、「インスタンス変数」、「オブジェクトの属性」、あるいは「オブジェクトのプロパティ」と呼ぶ
- 例：

```
from datetime import datetime # datetimeクラスを利用

cal = datetime()  # オブジェクトを作成

print( cal.year )  # year属性
```

日付時刻オブジェクト

- datetimeモジュールに入っている
  - **import** datetime
- calendarモジュールとtimeモジュールもある
  - **import** calendar, time

- datatimeモジュールのクラス
  - date　…　日付だけを表す
  - time　…　時間を表す
  - datetime　…　日付・時間を表す
  - timedelta　…　時間差（何時間か）を表す
  - tzinfo / timezone　…　時間帯を表す

日付・時間のためのクラス datetime

- オブジェクトの作り方
  - 指定された日時を表すオブジェクト
    - cal = datetime.datetime(西暦年, 月, 日)
  - 実行された瞬間の日時を表すオブジェクト
    - now = datetime.now( )
  - 指定した時間帯の瞬間の日時を表すオブジェクト
    - zone = timezone(timedelta(hours=0))
    - zonenow = datetime.now(tz=zone)
  - 指定した時間帯の日時に変換するメソッド（関数）
    - cal.astimezone(zone)

日付時間の指定

- 日時を指定したオブジェクトの作成
  - cal = datetime.datetime(西暦年, 月, 日, ...)
- オプションのパラメータ
  - *hour* = 時間（0〜23）
  - *minute* = 分（0 ~ 59）
  - *second* = 秒（0 ~ 59）
  - *microsecond* = マイクロ秒（0 ~ 999999）
- 例：

  datetime.datetime(2023, 11, 8)　# 2023年11月8日0時0分0秒

  datetime.datetime(2023, 11, 8, *hour*=16, *minute*=30, *second*=12)　# 2023年11月8日16時30分12秒

  datetime.datetime(2023, 11, 8, *microsecond*=567890) # 2023年11月8日0時0分0秒567890マイクロ秒

求めたい要素

- 以下のようにしてオブジェクトを取ってきている
  - cal = datetime.datetime(....)
- 日時は、分解して次のように求められる（整数値）
  - cal.year … 西暦の年
  - cal.month …月 1 ~12
  - cal.day … 月の中の日
  - cal.hour … 時間 0 ~ 23
  - cal.minute … 分 0 ~ 60
  - cal.second … 秒 0 ~ 60
  - cal.microsecond … マイクロ秒 1000000で1秒
  - cal.weekday() … 月曜日を0 〜 日曜日を6
  - cal.isoweekday() … 月曜日を1 〜 日曜日を7

strftime/strptime関数による文字列との変換

- datetimeクラスのオブジェクトは、strftime関数（メソッド）で、フォーマットされた文字列に変換することができる（C言語との互換性）
- 書式：オブジェクト.strftime(フォーマット文字列)
  → フォーマットされた文字列が返される
- strptime関数を用いて、文字列と変換用のフォーマット文字列から、datetimeクラスのオブジェクトを作成することができる
- 書式：datetime.strptime(文字列, フォーマット文字列)
  → datetimeクラスのオブジェクトが返される
- フォーマット文字列は、以下のような指定子を使うことができる
  - %y…西暦2桁
  - %Y…西暦4桁
  - %m…月2桁
  - %d…日2桁
  - %D…%m/%d/%yと同じ
  - %a…曜日を省略形の英語で
  - %A…曜日を英語で
  - %b…月の省略名
  - %B…月を英語で
  - %H…時間を24時間形式で
  - %I…時間を12時間形式で
  - %M…分（00〜59）
  - %S…秒（00〜59）
  - %p…AMあるいはPM
- 使用例：

```
cal = datetime( 2023, 12, 8, hour=23, minute=35, second=10 )
print( cal.strftime( "%Y/%b/%d %H:%M:%S" ) )
# 2023/Dec/08 23:35:10
aDate = datetime.strptime( "3/23/2023 19:05", "%m/%d/%Y %H:%M" )
# datetime.datetime(2023, 3, 23, 19, 5)
```

時間差の求め方

- datetimeモジュールのtimedeltaクラスのオブジェクトを利用する

```
from datetime import timedelta  # クラスを利用
```

- datetimeクラスのオブジェクト同士を引き算すると、timedeltaクラスのオブジェクトになる

```
day1 = datetime.datetime( 2020, 7, 7 )
day2 = datetime.datetime( 2000, 1, 3 )
delta = day1 - day2
```

- timedeltaクラスのオブジェクトでは、days属性で何日間か、secondsで属性で何秒間か（１日のうち）、microseconds属性で何マイクロ秒か（１秒間のうち）が求められる

```
print( delta.days,"日" )

print( delta.seconds // 3600, "時間" )

print( delta.seconds // 60 % 60, "分", delta.seconds % 60, "秒" )
```

時間差を作る

- timedeltaクラスのオブジェクトを作成する
  - timedelta(days=日数, seconds=秒数)
  - days、secondsは、省略しても良い。その場合は、0と仮定される
- 例：

```
cal = datetime.now( )
delta = timedelta( days=6, seconds=7461 )
print( cal + delta )
```

timedelta(seconds=3600*24*3+ 382) # secondsだけで1日を越える秒数を指定しても良い

→ datetime.timedelta(days=3, seconds=382) # と同じになる。

時間計測の仕方

- timeモジュールのtime関数を利用する
  - **import** time # timeモジュールを使う
  - *start_time* = time.time( ) # 計測開始時time関数で求める
  - *now* = time.time( ) # 現在の時間

•time関数の戻り値には、1970年1月1日0時0分0秒から経過時間（UTC: Universal Time Coordinated世界協定時間）が秒単位で入る。秒より小さい部分は小数として入る。

- 時間計測は、開始時の時間を引く
  - *delta* = *now* - *start_time* # 時間差を求める
  - *seconds* = *delta* / / 1 % 60 # 小数以下を削除 0〜59の間
  - *minutes* = *delta* / / 60 % 60 # 分 0〜59の間
  - *hours* = *delta* / / 3600 % 24 # 時間 0〜23の間
  - *days* = *delta* / / (3600 * 24) # 何日か

TextCalendar, Calendarクラス

- calendarモジュールのTextCalendarクラスを使う
  - **from** calendar **import** TextCalendar …ライブラリを使う準備
- cal = TextCalendar( )…まず、オブジェクトを作る、以下の関数は作られたオブジェクトに対して発行する
  - formatmonth(year, month, w=2)…その月のカレンダーを文字列で返す（wは日付の幅の文字数）
  - formatyear(year, w=2, m=3)…その年のカレンダーを文字列で返す（mは横に並べられる月の数）
  - prmonth(year, month)…その月のカレンダーを表示する
  - pryear(year)…その年のカレンダーを表示する
  - setfirstweekday(weekday) … 0だと月曜始まり、6だと日曜始まり
- calendarモジュール直属で、TextCalendarと同じ機能を持つ関数が用意されている
  - month(year, month, w=2)…文字列で返す
  - calendar(year, w=2, m=3)…文字列で返す
  - monthrange(year, month)…(その月の1日の曜日, その月の最終日)が返される
  - setfirstweekday(weekday)…何曜日始まりか
  - prmonth(year, month)…その月のカレンダー表示
  - prcal(year)…その年のカレンダーの表示

with文

- **with** オブジェクト：　ブロック
- ブロック内で、オブジェクトに対しての命令であることが仮定される、ただしそのオブジェクトのクラス定義で、__enter__および__exit__メソッドを持っているもののみ使える
- 動かない例：

```
with cal:
    print( cal.year, cal.month, cal.day )
```

- **with** オブジェクト **as** 省略名：　ブロック
- ブロック内で、オブジェクトを省略名で参照できる
- 動かない例：

```
with datetime.now( ) as n:
    print( n.year, n.month, n.day )
```

with文

- 動く例：ファイルから読込みをする

```
with open( "test.data" ) as f:
    text = f.read( )
    print( text )
    f.close( )
```

ファイルからの読込み

- f = open(ファイル名, モード)
  - モードは、"r"…読み、"w"…新規作成・クリア後上書き
  - "r+"…更新用に読み、"w+"…更新用に書込み、
  - "a"…追加書込み, "b"…バイナリモード
  - Windowsで標準の読込み指定がShift JIS（cp932）になっている場合があるので、そのときは、encoding="utf-8"を付ける必要がある
- f.read( )…テキスト全体を読み込む
- f.readlines( )…テキストを改行で区切って 1 行ずつのリストとして読み込む
- f.readline( )…テキストを 1 行ずつ読み込む
- f.close( )…読込み終了

ファイルへの書込み

- f.seek(バイトのオフセット)
  f.seek(バイトのオフセット, 起点)
- ファイル内の任意の場所に書込み（読込み）位置を移動する
- 起点は、0…ファイルの先頭、1…現在の位置、2…ファイルの最後、起点が1の場合は、オフセットの値は負の値でも良い、2の場合は通常負の値
- f.write(データ)
- ファイルの現在の位置に、データを書き込む

CSVファイルの読み書き

- csvのモジュールだと１次元リストとCSVファイルの１行データを直接読み書きできる。文字列データで、途中に空白が入ってもOKなのが特徴
- 読込み

```
import csv
matrix = [ ]
f = open( filename, "r" )
reader = csv.reader( f )
for row in reader:  matrix.append( row )
f.close( )
```

- 書込み

```
f = open( filename, "w" )
#  空行が入ってしまう場合は、newline=""を入れる
writer = csv.writer( f )
for row in matrix: writer.writerow( row )
f.close( )
```

インターネットからの読込み

- urllib.requestモジュールを利用する
- https でのアドレス指定の場合、sslのcertificateを回避する設定を入れる
- readで取り出した内容は、byte文字列になっているため、decode()関数でUnicodeとして解釈させる
  - 記述例

```
import urllib.request
import ssl
ssl._create_default_https_context = ssl._create_unverified_context

responce = urllib.request.urlopen( "https://www.sfc.keio.ac.jp")
content = responce.read( )
responce.close( )

lines = content.decode( ).splitlines()
for i, line in enumerate( lines, start=1 ):
    print (str( i ).zfill( 4 ), ":", line )
```

sqlite3を利用する

- コマンドベースの場合（Mac OS Xのみ）
- 「sqlite3 データベース名」で起動
  - 例：sqlite3 test.db
  - そのフォルダにtest.dbというファイルが生成される
- コマンド入力が出てくるので、コマンドを入れる
  - 例：.help　→　ヘルプ画面が表示
  - .quit　→　終了
  - .tables　→　データベース内のテーブル一覧
  - .open データベース名　→　そのデータベースを開く
  - .read SQLファイル名　→　そのファイルに書かれたSQLを実行する
  - select * from sqlite_master;　→　データベースの情報一覧
- sqlite3から使えるSQLコマンドについては、以下のWebページを参照
  - http://rktsqlite.osdn.jp
  - https://www.sqlite.org/lang.html

Windowsでsqlite3ツールを使う

- www.sqlite.orgにアクセス
- Downloadのタブをクリックする
- Windowsのsqlite-

Precompiled Binaries for Windows

sqlite-dll-win32-x86-3280000.zip (472.74 KiB)	32-bit DLL (x86) for SQLite version 3.28.0. (sha1: c2f99e749ce8b66b9da7cd793dc9b254c395f8c5)
sqlite-dll-win64-x64-3280000.zip (786.76 KiB)	64-bit DLL (x64) for SQLite version 3.28.0. (sha1: c59d10acabb1346a6426d0e54da985683157be2c)
sqlite-tools-win32-x86-3280000.zip (1.70 MiB)	A bundle of command-line tools for managing SQLite database files, including the command-line shell program, the sqldiff.exe program, and the sqlite3_analyzer.exe program. (sha1: 4063fe326243ab775a86c104fa77ac178f03976b)

- 圧縮ファイルをC:\Program files(x86)のフォルダの下に展開
- コマンドプロンプトで、以下のようにして起動

```
cd C:\Program files(x86)\sqlite3
sqlite3.exe  C:\Users\ユーザ名\test.db
```

python3からsqlite3を利用する

- pythonをインストールすれば、標準でsqlite3のライブラリがインストールされている
- データベースに接続して、カーソルを作る
  - 例：**import** sqlite3

    conn = sqlite3.connect("test.db")　# 接続

    cur = conn.cursor( )　# カーソルを作る
- カーソルに対して、execute( )関数を呼ぶと実行ができる
  - 例：cur.execute("select * from sqlite_master")

実行結果の受け取り

- sqlite3では、select文などのSQLの実行結果を、タプルのリストとして返してくる
- for文で 1 行ずつ受け取る
  - 例：**for** row **in** cur.execute("select * from sample"):

    print(row)

- なお、データの更新や追加、削除などテーブルに変更を加えた場合は、最後にSQLでcommitコマンドを発行する必要がある
  - 例：cur.execute("commit")

- cur.execute関数を使って実行するときは、実行結果が失敗するときに備えて、try except文に入れておく必要がある。

SQL（sqlite3）の主なコマンド（簡易版）

- テーブルを作成する
  - create table [if not exists] テーブル名 (
    カラム名　データ型 [, カラム名 データ型 ...])
  - データ型には、text, int, num, real以外にもISO標準のデータ型（smallint, char, varchar）なども使える。画像などのBLOB型の場合には、データ型に何も記述しない
- テーブルにデータを追加する
  - insert into テーブル名 values (データ値 [, データ値...])
- テーブルのデータを削除
  - delete from テーブル名 [where 条件]
- テーブルのデータを更新
  - update テーブル名 set カラム名 = 式 [where 条件]
- テーブルのデータを検索
  - select カラム名あるいは* from テーブル名 [where 条件]