第Q章 Python 発展編

前章はいわば Python の基本編であったが、この章では、Python の発展的な特徴やライブラリーについて説明する。ただし、ここでの基本と発展の分け方は恣意的であり、何らかの基準に沿っているわけではない。単に説明しやすそうな順番に並べているだけである。

また、Python は人工知能（機械学習）やデータ解析の分野で非常に人気が高い言語であるが、これらを紹介しだすとキリがないので、ここでは人工知能やデータ解析関係のライブラリーの紹介は除外する。

Q.1 ファイル入出力

Python でファイルを読み書きするには、`open` 関数を使う。`open` 関数は、ファイル名とモードを引数に取り、ファイルオブジェクトを返す。モードは、読み込みならば `'r'`、書き込みならば `'w'` を指定する。ファイルを閉じるときは、ファイルオブジェクトの `close` メソッドを呼び出す。

`open` 関数は他にもいくつかのキーワード引数を取ることができる。詳しい説明は `open` 関数のドキュメントで調べることができる。

ファイルを一行ずつ読み込むときはファイルオブジェクトの `readline` メソッドを使うことができる。ファイルに書き込むときは、ファイルオブジェクトの `write` メソッドを使う。そのほかのメソッドは、 Python チュートリアルの「入力と出力」で調べることができる。

以下の例は整数が記録されているファイルを読み込んで、カウントアップするプログラムである。

ファイル名: openclose.py

```
try:
    fin = open("counter.txt", "r")
    try:
        count = int(fin.readline())
    except ValueError:
        count = 0
    fin.close()
except FileNotFoundError:
    count = 0
count += 1
fout = open("counter.txt", "w")
fout.write(str(count))
fout.close()
print(f"カウンターの値: {count}")
```

with 文

with 文を使うと、ファイルオブジェクトを自動的に閉じることができる。ファイルの閉じ忘れは大きな事故になりかねないので、ファイルを開くときは `with` 文を使うことを強く推奨する。`with open("ファイル名", "モード") as f:` というカタチで開始して、ファイルオブジェクト *f* に対する操作をインデントして書く。

以下の例は、上と同じ処理を with 文を使って書き直したものである。

ファイル名: withexample.py

```
1  import os
2
3  if os.path.exists("counter.txt"):
4      with open("counter.txt", "r") as fin:
5          try:
6              count = int(fin.readline())
7          except ValueError:
8              count = 0
9  else:
10     count = 0
11
12 count += 1
13 with open("counter.txt", "w") as fout:
14     fout.write(str(count))
15
16 print(f"カウンターの値: {count}")
```

なお、この例では `FileNotFoundError` を避けるために、事前に `os.path.exists` という関数を使ってみた。この関数を使うためには、`import os` が必要である。

> **問 Q.1.1** 数値が並べられた CSV ファイルを読み込んで、各縦の列の合計を計算して出力するプログラムを作成せよ。

Q.2 タートルグラフィックス

Python には標準で `turtle` というタートルグラフィックスのライブラリーが備わっている。タートルグラフィックスはその名の通り、"亀" 🐢 を操って絵を描くというコンセプトに基づいている。タートルグラフィックスは簡単なプログラムでそれなりに複雑な絵を生成することができるので、初心者向けのプログラミングの題材として、よく取り上げられる。

タートルグラフィックスを使うには、まず `from turtle import *` という文でライブラリーをインポートする。以下にタートルに対する関数の一部を示す。

命令	意味
`forward(`*len*`)`	タートルを指定した距離 *len* だけ前進させる
`right(`*angle*`)`	タートルを指定した角度 *angle* だけ右に回転させる
`left(`*angle*`)`	タートルを指定した角度 *angle* だけ左に回転させる
`color(`*color*`)`	タートルの描く線の色を *color* にする
`penup()`	タートルのペンを上げ、移動中に線を描かなくする
`pendown()`	タートルのペンを下ろし、移動中に線を描くようにする

そのほかにも多くの関数があり、turtle のドキュメントに解説されている。

以下の例は、タートルグラフィックスを使って色を変えながら、ある規則で線分を描くプログラムである。

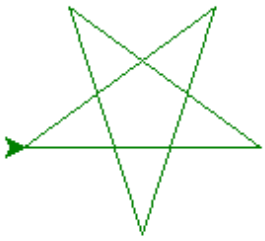
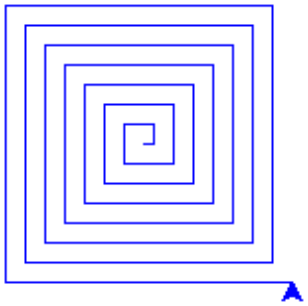
ファイル名: turtleexample.py

```
from turtle import *

colors = ['red', 'yellow4', 'green',
          'cyan4', 'blue', 'magenta4']
for i in range(60):
    color(colors[i % len(colors)])
    forward(120)
    right(95)

input("Press Enter to exit")
```

問 Q.2.1 次のような図形を描くプログラムを turtle を使って作成せよ。

(1)	(2)
	

その他の標準ライブラリーは、Python 標準ライブラリー (https://docs.python.org/ja/3/library/index.html) のドキュメントにまとめられている。正規表現や日付・時刻の処理、データベースやネットワーク関係のライブラリーなども含まれている。どのようなライブラリーが存在しているか目を通しておいてほしい。

Q.3 Python のオブジェクト指向

Python は**オブジェクト指向プログラミング**をサポートするが、Java のようになんでもかんでもクラスにしなければいけないという言語ではないので、自分でクラスを定義しなければいけないという状況は Java よりは格段に少ない。ほとんどの場合、既存のクラスを使いこなせれば十分であろう。このことを一応お断りしたうえで、Python のオブジェクト指向プログラミングを紹介していく。しかし、うまく使いこなせば、独自の型を定義しながら、その型に対して既存のライブラリーを活用することもできるだろう。

オブジェクト指向は、もともとシミュレーションのような分野で誕生した概念である。対象となるデータをオブジェクトと呼んでいるが、オブジェクトの状態を属性として表現し、オブジェクトの振る舞いをメソッドとして表現する。振る舞い（メソッド）の種類は固定されていて将来的に増えていかず、ただしオブジェクトの種類（クラス）が追加されていくような場合に有効である。ク

ラスが追加された場合にも、既存のクラスに対して定義されたメソッドを利用するコードはそのまま使えるので、ソフトウェアの再利用性が高い。ゲームや業務システムもシミュレーションの一種と言ってよいので、オブジェクト指向はこれらの分野において有用性が高い。一方で、オブジェクトの種類が固定されていて、振る舞い（関数）があとから追加されていくような場合には、オブジェクト指向は有効ではない。

オブジェクト指向の特徴としてよく挙げられるのは、**継承**、**動的束縛**、**カプセル化**の 3 つのキーワードである。以下ではこれらを説明していく。

Q.3.1 最初のクラス

上で述べたように、オブジェクト (object) はいくつかの属性 (attribute あるいは property) とメソッド (method) を持つ複合的なデータである。 Python では、まず**クラス** (class) を定義することでオブジェクトのひな型を作成する。クラスは `class クラス名:` という形のあとに、メソッドなどの定義をインデントして記述する。

ここでは、上で紹介した turtle ライブラリーと同じような動作をするクラスを例として定義していく。ただし、スペースの都合上、グラフィックスは使用せず、標準出力に描画命令っぽいものを出力する形で実装する。

ファイル名 MyTurtle/my_turtle.py

```python
import math

class MyTurtle:
  """SVG で描画する簡易タートル"""

  def __init__(self, x=0, y=0, angle=0):
    self.x = x
    self.y = y
    self.angle = angle
    self.pen_down = True

  def line_to(self, x, y):
    if self.pen_down:
      print(f"<line")
      print(f"  x1='{self.x:.2f}' y1='{self.y:.2f}'")
      print(f"  x2='{x:.2f}' y2='{y:.2f}' />")
    else:
      print(f"<!-- Moving to ({x:.2f}, {y:.2f}) -->")

  def forward(self, distance):
    angle_rad = math.radians(self.angle)
    dx = distance * math.cos(angle_rad)
    dy = distance * math.sin(angle_rad)
    x1 = self.x + dx
    y1 = self.y + dy
    self.line_to(x1, y1)
    self.x = x1
    self.y = y1

  def right(self, da):
    self.angle = (self.angle + da) % 360

  def left(self, da):
```

```
34      self.right(-da)
```

クラスの定義は `class` というキーワードのあとにクラス名、コロン（:）で開始する。このあとにインデントしてメソッドなどの定義を書いていく。（これまで原則として 4 字インデントしてきたが、クラス内では インデントが深くなりすぎるため、スペースの都合上、2 字のインデントに切り替えている。）

なお、Python の慣習では、クラス名は大文字で始めるキャメルケース (CamelCase) 🐫で書くことになっている。つまり、`my_turtle` でも `myTurtle` でもなく、 `MyTurtle` という名前が良い。

クラスの定義も、関数の定義と同じように最初に説明のための文字列 (docstring) をつけておくことができる。この文字列はコメントのようなもので実行時には何の影響も及ぼさないが、説明の役割を果たしている。この例では "SVG で描画する簡易タートル" という docstring がつけられている。

そして、このクラスには `__init__`, `line_to`, `forward`, `right`, `left` の 5 つのメソッドが定義されている。

クラスを定義すると、クラス名を関数のように使うことで**インスタンス**を生成することができる。インスタンスは具体例という意味である。以下が使用例である。

```
from my_turtle import MyTurtle

t = MyTurtle(50, 0, 90)
t.forward(100)
```

この `MyTurtle(50, 0, 90)` の部分がインスタンス生成である。インスタンス生成では、`__init__` というメソッドを裏で呼び出す。まず、新しい空のオブジェクトを生成し、それを `__init__` の第 1 引数とする。 `__init__` の第 2 引数以降はインスタンス生成に渡された引数（上の例では 50, 0, 90）をそのまま使用する。つまり、この例では新しく生成したオブジェクトの `x` 属性に 50 が、`y` 属性に 0 が、 `angle` 属性に 90 が、`pen_down` 属性に `True` がそれぞれ設定される。

> **注意:** `__init__` のような `__` で始まり、`__` で終わる名前のメソッドは、Python で特別な用途に使われるメソッドである。このような名前を独自に定義するメソッドや変数に使用してはいけない。

このように生成されたインスタンスでは、インスタンスの属性を**関数として**アクセスすると、インスタンスそのものを第 1 引数として追加して、クラスで定義された同じ名前の関数が呼び出される、という約束になっている。つまり、上の例では `t.forward(100)` とすると、`MyTurtle.forward(t, 100)` が呼び出されることになる。一般的に *`obj.method(...)`* というメソッドの呼び出しでは *`obj`* が属するクラスによって、実際に実行されるコードが振り分けられる。つまり `xxx.do_something(1, "abc")` と `yyy.do_something(1,`

"abc") という呼び出しがあったとき、同じ do_something というメソッドであっても、xxx と yyy のクラスが異なれば、実際に呼び出されるコードは一般的に異なることになる。

結局、このプログラムでは、`<line x1='50.00' y1='0.00' x2='50.00' y2='100.00' />` が出力される。

余談: Python では、メソッドを定義するときの第 1 引数の名前を self とすることが慣例になっている。なお、Java をはじめとするほとんどのオブジェクト指向言語では、メソッド内で self のような第 1 引数を明示的に指定する必要はない。また、同じインスタンスのメソッドや属性にアクセスする場合は、単に line_to, x, y のように属性名だけ書けばよい。しかし Python は self を明示する設計を選択したようである。

補足: なお、ここの例では、SVG 形式で描画命令を出力しているので、前後に適切な SVG のタグをつけて、SVG ファイルとして保存すれば、ブラウザでそのグラフィックスを表示することができる。以下のような header, footer 関数で前後を挟んでやればよい。

```
def header():
    print("<svg xmlns='http://www.w3.org/2000/svg'")
    print("  version='1.1' width='294mm' height='210mm'")
    print("  viewBox='0 0 294 210'")
    print("  preserveAspectRatio='xMidYMid'>")
    print("<g transform='translate(147, 105)'")
    print("  stroke='black' stroke-width='1'")
    print("  stroke-linecap='round'>")

def footer():
    print("</g>")
    print("</svg>")
```

Q.3.2 継承

1 個のクラスを定義するだけではオブジェクト指向の意味がないので、同じメソッドを持つクラスを作成する。通常、同じメソッドを持つクラスを定義するときは、**継承** (inheritance) と呼ばれる機能を使う。（ただし、Python では継承によらなくても同じメソッドを持つクラスを定義できる。）継承は、あるクラスをもとにして新しいクラスを定義することである。もとになるクラスを**スーパークラス** (superclass) と呼び、新しく定義するクラスを**サブクラス** (subclass) と呼ぶ。サブクラスはスーパークラスのメソッドをそのまま引き継ぐことができる。また、サブクラスでスーパークラスのメソッドをオーバーライド (override) することもできる。

以下の例では、MyTurtle クラスを継承して、ColorTurtle クラスを定義している。継承する場合は、`class クラス名(スーパークラス名):` という形でクラスの定義を開始する。なお、スーパークラスをコンマで区切って複数指定することもできる（**多重継承**）が、ここでは多重継承の実例の紹介までは立ち入らない。

ファイル名 MyTurtle/color_turtle0.py

```
from my_turtle import MyTurtle

class ColorTurtle0(MyTurtle):
  def __init__(self, x=0, y=0, angle=0,
               color="black"):
    super().__init__(x, y, angle)
    self.color = color

  def line_to(self, x, y):
    if self.pen_down:
      print(f"<line")
      print(f"  x1='{self.x:.2f}' y1='{self.y:.2f}'")
      print(f"  x2='{x:.2f}' y2='{y:.2f}'")
      print(f"  stroke='{self.color}' />")
    else:
      print(f"<!-- Moving to ({x:.2f}, {y:.2f}) -->")
```

この ColorTurtle0 クラスでは、MyTurtle クラスに色の情報を追加している。コードでは __init__ と line_to メソッドだけを定義している。super() はオーバーライドされたスーパークラスのメソッドを呼び出すための書き方である。x, y, angle の 3 つの引数はスーパークラスの __init__ メソッドに渡される。ColorTurtle0 では color というプロパティーだけを追加している。line_to メソッドは、色の情報を出力する必要があるので、スーパークラスの line_to メソッドを置き換えている。

お断り: MyTurtle と ColorTurtle0 のようにクラスの定義を細かく分けたのは、継承の説明に使いたいからであり、このようなクラスの設計が良いという意味ではない。実際には、このような単純なクラスなら一つにクラスにまとめてしまうのが良いだろう。

Q.3.3 動的束縛

さて、このように定義された ColorTurtle0 クラスの forward メソッドを呼び出してみる。ColorTurtle0 クラスでは forward メソッドを定義していないが、その場合、スーパークラスの MyTurtle クラスの forward メソッドを引き継いでいる。

```
from color_turtle0 import ColorTurtle0

ct = ColorTurtle0(x=50, y=0, angle=90, color="red")
ct.forward(100)
```

このプログラムを実行すると、`<line x1='50.00' y1='0.00' x2='50.00' y2='100.00' stroke='red' />` という出力が得られる。つまり、forward メソッド自体は ColorTurtle0 ではオーバーライドされていないが、forward メソッドの内部で呼び出される line_to メソッドは ColorTurtle0 でオーバーライドされたほうを呼び出している。当たり前と思うかもしれないが、これがオブジェクト指向で肝腎な点である。

このような振る舞いを**動的束縛** (dynamic binding) と呼ぶ。`ColorTurtle0` クラスは、`MyTurtle` クラスより後に定義されるので、`MyTurtle` の `forward` メソッドが定義されているとき、`ColorTurtle0` クラスの `line_to` メソッドはまだ定義されていない。つまり `forward` メソッドが定義されたときに（静的に）わかっている `MyTurtle` クラスの `line_to` メソッドではなく、実行時に（動的に）わかるほうの `line_to` メソッドの実装が選択される、という意味である。

> **詳細:** プログラムの実行前にわかる性質のことを一般に**静的** (static) と呼び、プログラムの実行中にわかる性質のことを一般に**動的** (dynamic) と呼ぶ。

動的束縛は、オブジェクト指向のもっとも重要な特徴の一つである。

> 問 **Q.3.1** `ThickTurtle` クラスは `MyTurtle` クラスを継承して、線の太さを指定できる。`x, y, angle, pen_down` に加えて新しい属性 `width` を持っている。インスタンス生成のときには `x, y, angle` と `width` の初期値を引数として指定できるようにする。`line_to` メソッドも再定義されていて、線の太さを指定する `stroke-width` 属性を出力する。例えば、`width` が 2.71 のときは `<line x1='50.00' y1='0.00' x2='50.00' y2='100.00' stroke-width='2.71' />` のように出力する。この `ThickTurtle` クラスを定義せよ。

Q.3.4 カプセル化

しかし、この `ColorTurtle0` クラスの定義には重大な欠点がある。`color` という属性になんでも設定できてしまうので、`ct.color = "led"` のようにあり得ない色名を設定してしまうこともできる。

そこで `set_color` というメソッドを定義して色名をチェックすることにする。

ファイル名 MyTurtle/color_turtle1.py

```python
from my_turtle import MyTurtle

class ColorTurtle1(MyTurtle):
  _valid_colors = ["black", "red", "green", "yellow",
                   "blue", "magenta", "cyan", "white"]

  def __init__(self, x=0, y=0, angle=0,
               color="black",):
    super().__init__(x, y, angle)
    self.set_color(color)

  def line_to(self, x, y):
    if self.pen_down:
      print(f"<line ")
      print(f"  x1='{self.x:.2f}' y1='{self.y:.2f}'")
      print(f"  x2='{x:.2f}' y2='{y:.2f}'")
      print(f"  stroke='{self.get_color()}' />")
    else:
      print(f"<!-- Moving to ({x:.2f}, {y:.2f}) -->")

```

```
21    def get_color(self):
22      return self._color
23
24    def set_color(self, c):
25      if c in self._valid_colors:
26        self._color = c
27      else:
28        self._color = "black"
```

この `set_color` メソッドでは、引数に渡された色名が有効な色名のリスト (`_valid_colors`) に含まれているかどうかをチェックし、含まれていればその色に設定し、含まれていなければデフォルトの色（黒）に設定する。だから、`ct.set_color("led")` のようなことをしても、色は黒になる。

しかし、せっかく `set_color` のようなメソッドを用意しても、直接属性をアクセスされては意味がない。そこで Python では属性の名前の前にアンダースコア（_）をつけることで、その属性がクラスの内部でのみ使用されることを主張することができる。つまり、属性名やメソッド名の最初のアンダースコアはクラスの外部で使用しないでくれ、使用してあとで問題が起きても知らないよ、という意味である。

> **詳細:** 他のオブジェクト指向言語では、隠したい属性をクラスの外部からアクセスするとコンパイル時にエラーになるように設計されていることが多い。しかし、Python では、あくまでも慣用として変数名の前にアンダースコアをつけることで、クラスの外部からアクセスしないでくれと主張するだけにとどめている。なお、名前の最初にアンダースコアを 2 個つける（そして、最後はアンダースコア 2 個ではない）と、別の意味が生じるが、ここではその詳細には立ち入らない。

このように、クラスの実装の詳細を隠すことを**カプセル化** (encapsulation) と呼ぶ。カプセル化は、オブジェクト指向のもう一つの重要な特徴である。

なお、`_valid_colors` のようにクラスの中で宣言されている変数は、クラス変数といって、クラスのすべてのインスタンスで共有される変数になる。

> **問 Q.3.2** 先に定義した `ThickTurtle` クラスに対しても、線の太さを設定する `set_width` メソッドと、線の太さを取得する `get_width` メソッドを定義せよ。また、`set_width` メソッドでは設定できる線の太さを 0 以上 10 以下の実数に制限する（負の値を指定すれば 0 に、 10 を超える値を指定すれば 10 になるようにする）こと。そして、線の太さを直接属性にアクセスすることを防ぐために、線の太さを表す属性の名前を変更すること。

Q.3.5 プロパティー

上の例では `set_color` と `get_color` というメソッドを定義して、直接属性にアクセスすることを防いでいるが、やはり `ct.color` のような記法で属性にアクセスできるほうがプログラムを簡潔に保てるだろう。そこで、Python では **プロパティー** (property) といって `@property` と `@name.setter` というデコレー

ターを使うことで、属性にアクセスするような記法でメソッドを呼び出すことができる。以下が、 `@property` と `@color.setter` を使って書き換えたプログラムである。

ファイル名 MyTurtle/color_turtle.py

```
from my_turtle import MyTurtle

class ColorTurtle(MyTurtle):
  _valid_colors = ["black", "red", "green", "yellow",
                   "blue", "magenta", "cyan", "white"]
  def __init__(self, x=0, y=0, angle=0,
               color="black"):
    super().__init__(x, y, angle)
    self.color = color

  def line_to(self, x, y):
    if self.pen_down:
      print(f"<line ")
      print(f"  x1='{self.x:.2f}' y1='{self.y:.2f}'")
      print(f"  x2='{x:.2f}' y2='{y:.2f}'")
      print(f"  stroke='{self.color}' />")
    else:
      print(f"<!-- Moving to ({x:.2f}, {y:.2f}) -->")

  @property
  def color(self):
    return self._color

  @color.setter
  def color(self, c):
    if c in self._valid_colors:
      self._color = c
    else:
      self._color = "black"
```

デコレーターは関数の定義を修飾・変換するためのものだが、ここではこれ以上詳細には立ち入らない。

これで `ct.color = "led"` のようなことをしても、25 行めからのメソッドが呼び出され、色は黒 (black) に設定される。

> **問 Q.3.3** 先に定義した `ThickTurtle` クラスに対しても、線の太さを設定する `width` プロパティーを定義せよ。 やはり、同様に負の値を代入しようとすれば 0 に、 10 を超える値を代入しようとすれば 10 になるようにすること。

Q.3.6 静的型付け OOP 言語との相違

Python はオブジェクト指向のために大掛かりな仕組みを導入しておらず、ここまでだけですべてではないが、ほとんどの仕組みの説明が尽くされている。また、Python は実行前に型チェックを行わないので、他のオブジェクト指向言語と比べると、ある意味柔軟な設計が可能である。

例えば、クラスにあとでメソッドを追加することも可能である。もともとの `MyTurtle` クラスには `set_color`メソッド、 `get_color` メソッドはないので、次のようなプログラムはメソッドが見つからなくてエラーになる。

```
from my_turtle import MyTurtle

t = MyTurtle(50, 0, 90)
t.set_color("red")
```

しかし、`MyTurtle` クラスに `set_color` メソッド（と `get_color` メソッド）を追加することは可能である。上の `t.set_color("red")` を実行する前に、以下のようなコードを追加しておけばよい。

```
def turtle_set_color(self, color):
    pass

def turtle_get_color(self):
    return "black"

MyTurtle.set_color = turtle_set_color
MyTurtle.get_color = turtle_get_color
```

なお、プロパティーにしたければ（`t.color = "red"` のように書きたければ）最後の 2 行は次のように書く。

```
MyTurtle.color = property(turtle_get_color,
                          turtle_set_color)
```

ここでは、`set_color` は与えられた色を無視して、常に黒になるような実装にしているが、他の実装（例えば、`line_to` メソッドを書き換えてしまうこと）も可能である。

これは Java のような静的な型付けを行うオブジェクト指向言語では、通常実現できないことである。 もちろん、これは危険性を伴い、注意深く実装しないと、実行時にエラーとなったり、 予期しない動作となったりする可能性が高い。また、静的な型付けの言語と比較すると、 メソッドの振り分けの効率化に限界ができてしまうだろう。 しかし、うまく使えば Java のような静的な型付けを行うオブジェクト指向言語では実現できない柔軟性を得ることができる。

> **詳細:** 例えば、Kotlin の**拡張関数** (extension function) は、あとからクラスにメソッドを追加できるように見えるが、実際には、拡張関数はクラスのメソッドとは異なり、動的束縛をしない。つまり、コンパイル時にわかっている型でどのメソッドが呼び出されるかが決まってしまう。

Q.4 仮想環境とライブラリー

ここからは、Python 処理系に最初からは入っていないライブラリーを使用するので、必要なライブラリーをインストールする必要がある。

なお、ライブラリーをインストールするときは `venv` を使って**仮想環境** (virtual environment) を作成することを推奨する。 仮想環境を使わず、システム全体にライブラリーをインストールすると、例えばプロジェクト A では必要なライブラリーのバージョンが 1.2.3 で、プロジェクト B では同じライブラリーのバージョン 2.1.4 が必要なときに、どちらかのプロジェクトしか動かなくなってしま

う。 仮想環境を使うと、プロジェクトごとに異なるライブラリーのバージョンを管理できるようになる。

まず、プロジェクト用に新しいディレクトリーを作成し、そこに移動してから仮想環境を作成する。 仮想環境を作成するには、まず `python -m venv 仮想環境名` とする。これで、*仮想環境名* というディレクトリーが作成される。仮想環境名はなんでも良いのだが、 `.venv` が良く使われるようである。このディレクトリーの中に、Python の実行ファイルやライブラリーが格納される。

```
PS C:\…> mkdir MyProject
PS C:\…> cd MyProject
PS C:\…\MyProject> python -m venv .venv
```

仮想環境で作業をするときには、次のコマンドで仮想環境を有効にする。

```
PS C:\…\MyProject> .\.venv\Scripts\Activate
```

なお、上記は Windows の場合で、Linux の場合のパスは `.venv/bin/activate` になる。なお、仮想環境を有効にすると、プロンプトの最初に仮想環境名が追加されるはずである。

この仮想環境が有効になった状態で、ライブラリーをインストールしてプログラムを作成する。例えば NumPy をインストールするには、次のようにする。

```
(.venv) PS C:\…\MyProject> python -m pip install numpy
```

注意: `pip` をコマンドとして使用して、 `pip install numpy` とすることも可能である。ただしこうすると、誤ったバージョンの `pip` を使ってしまうことがあり、 `python -m pip install numpy` とするほうが安全であるとされている。

しかし、本資料ではスペースの都合上、以降では前者の `pip` をコマンドとして使う形で紹介していく。

なお、 `pip freeze` とすると、現在インストールされているパッケージの一覧が表示される。この一覧を `requirements.txt` というファイルに保存 `pip freeze > requirements.txt` しておくと、他のコンピューターでライブラリーのバージョンも含めて同じ環境を再現することができる。

この requirements.txt を使って、別の環境で同じパッケージをインストールするには、 `pip install -r requirements.txt` とする。

仮想環境での作業が完了したら、仮想環境を解除するために、次のコマンドを実行する。

```
(.venv) PS C:\…\MyProject> deactivate
PS C:\…\MyProject>
```

プロンプトから仮想環境の名前が消えるはずである。

Q.5 Jupyter Notebook

Jupyter Notebook は**ノートブック形式**という、プログラムのソースコードや、整形したテキスト、数式、グラフィックスなどを含むドキュメントを作成するためのウェブアプリケーションである。別の言い方をすると Web ブラウザーから実行できる、インタラクティブでグラフィカルな開発環境 (REPL, Read-Eval-Print Loop) である。 Jupyter Notebook は Python 以外の言語からでも使用できるが、Python 上のものが一番よく知られている。

Jupyter Notebook には JupyterLab という、より高機能な後継バージョンもあるが、ここではまとめて紹介する。

Q.5.1 Jupyter Notebook のインストール

仮想環境を有効にした状態で、次のコマンドでインストールする。

```
(.venv) PS C:\Users\…> pip install notebook
```

次のコマンドで起動する。

```
(.venv) PS C:\Users\…> jupyter notebook
```

これで Web ブラウザーが起動して、Jupyter Notebook の画面が表示される。最初は “New” のドロップダウンメニューから “Python 3 (ipykernel)” を選択して新しいノートブックを作成する。ノートブックの画面が表示されるので、セルにコードやマークダウンなどを入力することができる。入力して Shift + Enter キーを押すと、セルのコードが実行される。以下にノートブックの入力例を示す。

In [1]:
```
1 + 2 + 3
```

Out[1]:
```
6
```

In [2]:
```
def factorial(n):
    if n <= 0:
        return 1
    else:
        return n * factorial(n - 1)
```

In [3]:
```
factorial(100)
```

Out[3]:
```
933262154439441526816992388562667004907159682643
816214685929638952175999932299156089414639761565
182862536979208272237582511852109168640000000000
00000000000000
```

ノートブックは保存しておくと、あとで開いて編集・実行することができる。Jupyter Notebook のファイルは `.ipynb` という拡張子を持つ。

なお、作業が終了したら、“File” から “Shut Down” を選択して Jupyter Notebook を終了する。

JupyterLab の場合は、次のコマンドでインストールする。

```
(.venv) PS C:\Users\…> pip install jupyterlab
```

次のコマンドで起動する。

```
(.venv) PS C:\Users\…> jupyter lab
```

これで Web ブラウザーが起動して、JupyterLab の画面が表示される。Launcher の画面が表示されるので、 Python 3 のアイコンをクリックすると、Python のノートブックが起動する。ここに、コードやマークダウンなどを入力することができる。

作業が終了したら、“File” から “Shut Down” を選択して JupyterLab を終了する。

ちなみに残念ながら turtle は Jupyter Notebook ではうまく動作しない（別のウインドウに描画してしまい、また終了時に不具合が起きやすい）が、代わりに ColabTurtlePlus というライブラリーを使うと Jupyter Notebook 上でタートルグラフィックスを作成することができる。以下のコマンドでインストールすることができる。

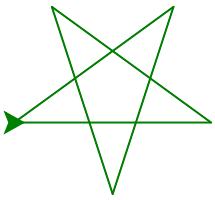
```
(.venv) PS C:\Users\…> pip install ColabTurtlePlus
```

ColabTurtlePlus の実行例を以下に示す。（この例では Jupyter Notebook 内 から `pip` を呼び出すときに ! をつけてシェルコマンドを実行している。）

```
In [1]: !pip install ColabTurtlePlus
```

```
Collecting ColabTurtlePlus
  Using cached ColabTurtlePlus-2.0.1-py3-none-any.
whl.metadata (10 kB)
Using cached ColabTurtlePlus-2.0.1-py3-none-any.wh
l (31 kB)
Installing collected packages: ColabTurtlePlus
Successfully installed ColabTurtlePlus-2.0.1
```

```
In [6]: from ColabTurtlePlus.Turtle import *
clearscreen()
setup(300, 150)
color("green")
for _ in range(5):
  forward(100)
  left(144)
```



Q.5.2 VS Code からの Juypter Notebook の実行

VS Code からも Jupyter Notebook を実行することができる。セルに入力するときに VS Code のコード補完 (IntelliSense) 機能を使えるので便利である。詳しいことは Jupyter Notebooks in VS Code というページに紹介されているが、ここでは一部のみ抜粋して説明する。

VS Code から Jupyter Notebook を実行するには、まず VS Code の拡張機能の “Jupyter” (by Microsoft) をインストールする。また、Jupyter Notebook あるいは JupyterLab 自体は VS Code の外で（仮想環境を使って）インストールしておく。（実際には ipykernel というライブラリーだけインストールしておけば良いようだが、VS Code 外からも Jupyter Notebook を実行できるようにするために、Jupyter Notebook あるいは JupyterLab をインストールしておくのがおすすめである。）

以上がインストールできていれば、実行の準備ができているので、VS Code のメニューから “View” → “Command Palette” を選択し、“Create: New Jupyter Notebook” を選択する。これで新しいノートブックが作成される。次に、右上の “Select Kernel” をクリックして、仮想環境（`.venv (Python 3.XX.X) .venv/Scripts/python.exe` のような項目があるはず）を選択する。

以降は、ブラウザーからと同じようにノートブックを操作できるはずである。

Q.6 NumPy

NumPy は、Python の数値計算ライブラリーである。特にベクトルや行列のようなデータを多次元配列を使って効率的に計算できるように設計されている。

インストールは以下のコマンドで行う。

```
(.venv) PS C:\Users\…> pip install numpy
```

ただし、後述の Matplotlib をインストールするときは、依存関係により NumPy も一緒にインストールされるので、NumPy だけをわざわざインストールする必要はない。

NumPy に用意されている関数は膨大であるため、詳しい使用法はドキュメンテーション、とくに API reference を調べてほしい。

Q.7 Matplotlib

Matplotlib は、Python のグラフィックスライブラリーである。 Jupyter Notebook 上で使用すると、ノートブック上にグラフを描画することができる。Python は他にも Plotly, Bokeh などのグラフィックスライブラリーがある。それぞれ特徴があって使い分けられているが、Matplotlib は以前から広く使われており、ドキュメントも充実している。

インストールは以下のコマンドで行う。

```
(.venv) PS C:\Users\…> pip install matplotlib
```

Matplotlib の Jupyter Notebook での実行例を以下に示す。(グラフ自体は Matplotlib の “Getting started” のページの例そのものである。)

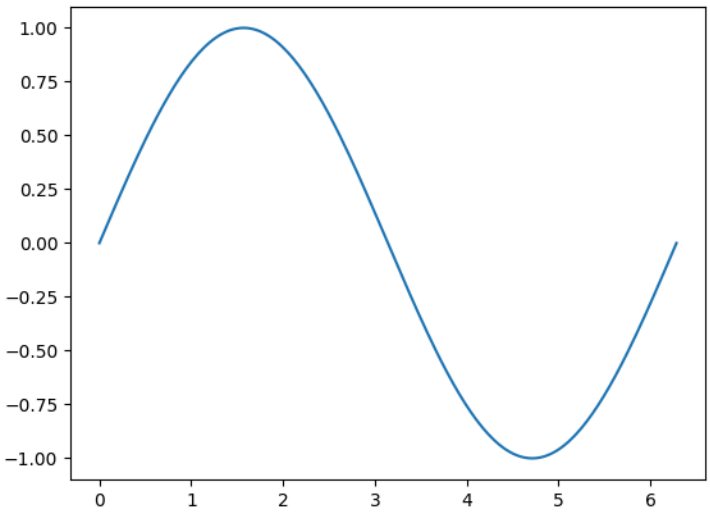
In [1]:
```
!pip install matplotlib
```

```
Collecting matplotlib
  Downloading matplotlib-3.10.3-cp313-cp313-win_am
d64.whl.metadata (11 kB)
:（途中略）
Successfully installed contourpy-1.3.2 cycler-0.1
2.1 fonttools-4.58.4 kiwisolver-1.4.8 matplotlib-
3.10.3 numpy-2.3.1 pillow-11.2.1 pyparsing-3.2.3
```

In [2]:
```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2 * np.pi, 200)
y = np.sin(x)

fig, ax = plt.subplots()
ax.plot(x, y)
plt.show()
```



ここで `import ～ as ～` というカタチでライブラリーをインポートしているが、これはライブラリーの名前が長いときに、短い名前で使えるようにするためのものである。例えば `import matplotlib.pyplot as plt` とすると、Matplotlib の `pyplot` モジュールを `plt` という短い名前で使えるようになる。

Matplotlib の関数もやはり膨大であるため、詳しい使用法はドキュメンテーションを参照してほしい。

> **問 Q.7.1**
>
> - $y = \sum_{k=1}^{7} \frac{\sin(kx)}{k}$ のグラフを描け。

- `xs = [100, 200, 300, 400, 500]`、`ys = [31, 29, 25, 28, 23]` のように x 座標、y 座標を与えたときに折れ線グラフを描く方法を調べよ。
- `labels = ["徳島", "香川", "愛媛", "高知"]` 、 `values = [679, 911, 1262, 647]` のようなデータから棒グラフを描く方法を調べよ。

Q.8 Web アプリケーション (Flask)

Web アプリケーションは、サーバー側で動作し、Web ブラウザーを通じて利用されるアプリケーションである。 Python には Django や FastAPI などの Web アプリケーションフレームワークもあるが、ここでは、Flask という軽量な Web アプリケーションフレームワークを紹介する。

インストールは以下のコマンドで行う。

```
(.venv) PS C:\Users\…> pip install Flask
```

Flask を使ったとても簡単な Web アプリケーションの例を示す。Web ページは HTML を返す関数として実装される

ファイル名 MyFlask/app.py

```python
from flask import Flask

app = Flask(__name__)

@app.route("/")
def greet():
  """時間に応じて挨拶を返す関数"""
  from datetime import datetime
  now = datetime.now()
  current_hour = now.hour
  current_sec = now.second
  color = "blue" if current_sec % 10 <= 5 else "red"

  if current_hour < 12:
    return f"<p style='color: {color};'>おはよう</p>"
  elif current_hour < 18:
    return f"<p style='color: {color};'>こんにちは</p>"
  else:
    return f"<p style='color: {color};'>こんばんは</p>"

```

これを実行するときは、`flask run` とする。すると、デフォルトでは http://127.0.0.1:5000/ にアクセスするとページが表示される。

`@〜.route("〜")` はページを配備する URL のパスを示すデコレーターである。このアプリケーションでは、`/` にアクセスすると、`greet` 関数が呼び出され、時刻に応じた挨拶の言葉が表示される。もしデコレーターが `@app.route("/hello")` であった場合、`http://127.0.0.1:5000/hello` にアクセスすると、 `greet` 関数が呼び出される。

問 Q.8.1 上記のプログラムを 3 色以上を使って、秒によって色を変えるように変更せよ。

もう一つ、POST を使ってフォームからの入力を受け付ける簡単な Web アプリケーションの例を示す。 POST はフォームなどからデータを受け取るときに使われる HTTP リクエストの種類である。一方、データを受け取らないか、クエリ文字列という URL の ? 以降に追加される文字列でデータを受け取るときは GET が使われる。

ファイル名 MyFlask/colors.py

```
from flask import Flask, request

app = Flask(__name__)

# https://www.colordic.org/w から抜粋
cdict = {
  "小豆": 0x96514d, "萌葱": 0x6e54, "茄子紺": 0x824880,
  "蜜柑": 0xf08300, "群青": 0x4c6cb3, "浅葱": 0x00a3af }

form_str = """
  <form method="POST">
    <input type="text" name="color_str"
           size="20" placeholder="色の名前を入力">
    <input type="submit" value="送信">
  </form>"""

@app.route("/", methods=["GET", "POST"])
def colors():
  """色のリストを返す関数"""
  if request.method == "GET":
    return form_str
  str = request.form.get("color_str", "")
  if str:
    ss = str.split(" ")
    color_list = "\n<ul>\n"
    for s in ss:
      color_list += "<li style='color: "
      color_list += f"#{cdict.get(s, '000000'):06x}"
      color_list += f"'>{s}</li>\n"
    color_list += "</ul>\n"
    return (form_str + color_list)
  return form_str
```

HTTP メソッドが POST のとき、`request.form.get` でフォームに入力された文字列を得ることができる。

これを実行するときは、`flask --app colors run` とする。（ファイル名が app.py のときは `--app` オプションを指定する必要はない。それ以外のファイル名のときは `--app` オプションを指定する。）

問 Q.8.2 GET でアクセスされたときは、品名と単価の表と、その個数を入力してもらうためのフォームを表示し、 POST でアクセスされたときは、そのフォームから入力を受け取って、見積書のようにテーブルとして整形して合計金額を表示するプログラムを作成せよ。なお、商品の単価などのデータは（本来は

> データベースから取得するものだが）プログラム中に辞書型の定数として埋め込んでおいてよい。

Flask の詳しい使い方は、Flask Documentation を参照してほしい。QuickStart のページだけでも多くの例が収録されている。セッションも容易に扱うことができるし、 HTML や画像ファイルなどの静的ファイルを提供する方法や、Jinja というテンプレートエンジンの使い方なども紹介されている。

> **問 Q.8.3** Flask で静的な HTML ファイルとテンプレートエンジンを利用する方法を調べ、上記の例題や問を Flask のテンプレートエンジンを使って実装せよ。

Q.9 Beautiful Soup

Beautiful Soup は、HTML 文書の解析を行うためのライブラリーである。ウェブスクレイピングによく使用される。 HTML の構造を理解して、特定の要素を抽出したりすることができる。

インストールは以下のコマンドで行う。

```
(.venv) PS C:\Users\…> pip install beautifulsoup4
```

詳しい使い方は 公式ドキュメントを参照してほしい。

Q.10 OpenPyXL

OpenPyXL は、Excel ファイルを読み書きするためのライブラリーである。 Excel ファイルをプログラムから操作することができる。

インストールは以下のコマンドで行う。

```
(.venv) PS C:\Users\…> pip install openpyxl
```

詳しい使い方は 公式ドキュメントの API Documentation を参照してほしい。

Q.11 その他の Python ライブラリー

Python の人気のライブラリーのまとめサイトとしては Awesome Python (`https://github.com/vinta/awesome-python`) が知られている。膨大な数のライブラリーがリストアップされているので、ライブラリーを探すときは、まずはこのサイトを見るとよい。