

プログラミング初級 (Python)

モジュールとパッケージ

早稲田大学グローバルエデュケーションセンター

プログラムの部品化と利活用

プログラムの美德については、ほとんどの方がご存知だろう。
ぐらたら、せっかち、思い上がりである。

— ラリー・ウォール

プログラムの部品化

- **再利用性が向上:**
モジュール化することで、特定の機能や機能のグループを別々のモジュールに分割できる。これにより、同じ機能を他のプログラムで再利用することが容易になる。他のプロジェクトや同じプロジェクト内の別の部分で同じモジュールを使用できる。
- **メンテナンスの容易性が向上:**
プログラムをモジュールに分割すると、変更や修正が必要な場合、関連するモジュールだけを修正すればよくなる。これにより、コード全体の理解が容易になり、メンテナンスが効率的に行える。
- **可読性が向上:**
モジュールを使用すると、コードの読みやすさが向上する。プログラム全体が一つのファイルに収まるよりも、小さなモジュールに分割された方が理解しやすくなる。各モジュールは特定の機能や役割を果たし、単一の責務を持つようになる。
- **グループ開発の容易性が向上:**
プログラムがモジュール化されている場合、異なる開発者やチームが同時に異なるモジュールに取り組むことができる。各モジュールが独立しており、相互に影響を与えずに開発が進められるため、大規模なプロジェクトやグループ開発が容易になる。
- **テストの容易性が向上:**
モジュールは単体でテストすることが可能。各モジュールの機能が独立しているため、テストケースを設計しやすく、バグを見つけやすくなる。
- **名前空間整理の容易性が向上:**
モジュールを使用すると、変数や関数などの名前空間が整理される。モジュール内で定義された名前は、そのモジュール内でのみ有効であり、他のモジュールとの名前の競合を避けることができる。

モジュールとパッケージ

- プログラムを部品として見たとき、それを**モジュール**と呼ぶ。
- 基本的には、Pythonプログラムを単一の**ファイル**として保存すると、それがモジュールとして利用可能になる。
- あるPythonプログラムからモジュールを読み込むには**import文**を書く。
- モジュールをまとめた**ディレクトリ**のことを**パッケージ**と呼ぶ。
- **from文**の記述によって他のPythonプログラムの値や関数を利用可能。
- 全ての構文に**as文**で、alias(別名)を利用可能。記述量を削減できる。

構文

```
import module
```

構文

```
import package.module
```

構文

```
import package.subpackage.module
```

構文

```
import module as alias
```

as文の
例は抜粋。

構文

```
import package.module as alias
```

構文

```
from package import module as alias
```

構文

```
from module import method
```

構文

```
from package import module
```

構文

```
from package.module import method
```

構文

```
from package.subpackage import module
```

構文

```
from package.subpackage.module import method
```

モジュールの作成とインポートの方法：import文のみの利用

```
1 author = "Pythonista"
2 def triangle(symbol, num):
3     """moduling a triangle with given symbol. """
4     for i in range(num):
5         for j in range((num - i) - 1):
6             print(end=" ")
7         for j in range((i + 1)):
8             print(symbol, end=" ")
9     print()
```

module.py

```
1 import module
2
3 print(module.author)
4 module.triangle('*', 5)
```

main.py

カレントディレクトリはホームディレクトリ。



```
> dir Desktop
(略)
```

OneDrive連携している場合は`dir OneDrive¥デスクトップ`。
保存場所をデスクトップ以外にしている場合はこの限りではない。

```
2023/11/16 18:41
```

```
86 main.py
```

```
2023/11/16 18:42
```

```
284 module.py
```

```
(略)
```

カレントディレクトリはホームディレクトリ。



```
% ls Desktop
module.py
```

保存場所をデスクトップ以外にしている場合はこの限りではない。

```
main.py
```

作業のポイント

- これらのファイルを**同じディレクトリ**に置く。
- これらのファイルが同じディレクトリにあることを**OSのコマンド**で確認。

モジュールを利用したPythonプログラムを実行



```
> python Desktop¥main.py
Pythonista
  *
 * *
* * *
* * * *
* * * * *

> dir Desktop
(略)
2023/11/16  18:41                86 main.py
2023/11/16  18:42            284 module.py
2023/11/16  18:43      <DIR>      __pycache__
(略)
```



```
% python3 Desktop/main.py
Pythonista
  *
 * *
* * *
* * * *
* * * * *

~ % ls Desktop
__pycache__  main.py      module.py
```



`__pycache__` (Pythonキャッシュ) は、Pythonのバイトコード (.pyc) ファイルがキャッシュされるディレクトリ。バイトコードはソースコードとバイナリコード間の中間表現に相当する。Pythonのコードは通常、ソースコード (.py ファイル) からバイトコードに変換されて実行される。バイトコードは、ソースコードよりも効率的に実行することができる形式。

Pythonは、コードの再コンパイルを避けるためにバイトコードを生成し、生成されたバイトコードは `__pycache__` ディレクトリに保存される。これにより、同じソースコードが再度実行される際に、バイトコードが再生成されるのを防ぐことができ、実行速度が向上する。

例えば、`module.py` という名前のPythonプログラムを実行した場合、そのバイトコードは `__pycache__` ディレクトリに保存される。実際には、`__pycache__` ディレクトリ内には `module.cpython-<version>.pyc` といった形式のファイルが作成される。

パッケージにあるモジュールを利用する例（準備）



```
> mkdir Desktop¥package
> dir Desktop
(略)
2023/11/16  18:41                86 main.py
2023/11/16  18:42             284 module.py
2023/11/16  19:15          <DIR>      package
2023/11/16  18:43          <DIR>      __pycache__
(略)
> copy Desktop¥module.py Desktop¥package
1 個のファイルをコピーしました。
> dir Desktop¥package
(略)
2023/11/16  18:42             284 module.py
(略)
```

この後、元のmodule.pyとコピーしたmodule.py
の区別がつくように**内容を編集**する。



```
% mkdir Desktop/package
% ls Desktop
__pycache__  main.py      module.py    package
% cp Desktop/module.py Desktop/package
% ls Desktop/package
module.py
```

この後、元のmodule.pyとコピーしたmodule.py
の区別がつくように**内容を編集**する。

from文とimport文を利用したモジュールのインポート

```
1 from package import module
2
3 print(module.author)
4 module.triangle('*', 5)
```

main.py



```
I> python Desktop¥main.py
```



```
I% python3 Desktop/main.py
```

実行結果の例

Pythonista

```
  *
 * *
* * *
* * * *
* * * * *
```

保存のスタイルが異なるだけで同じモジュールを実行しているので実行結果は変わらない。

疑問

- 次のようなプログラムの場合、どうなる？
- モジュールの保存場所が違うものの、モジュール名が同じ。

```
1 import module
2 from package import module
3
4 print(module.author)
5 module.triangle('*', 5)
```

main.py

```
1 author = "Pythonista"
2 def triangle(symbol, num):
3     """moduling a triangle with given symbol. """
4     for i in range(num):
5         for j in range((num - i) - 1):
6             print(end=" ")
7             for j in range((i + 1)):
8                 print(symbol, end=" ")
9             print()
10    print("This is a module.")
```

編集部分。

module.py

```
1 author = "Pythonista"
2 def triangle(symbol, num):
3     """moduling a triangle with given symbol. """
4     for i in range(num):
5         for j in range((num - i) - 1):
6             print(end=" ")
7             for j in range((i + 1)):
8                 print(symbol, end=" ")
9             print()
10    print("This is a module from a package.")
```

編集部分。

package/module.py

同じモジュール名が存在する場合

```
1 import module
2 from package import module
3
4 print(module.author)
5 module.triangle('*', 5)
```

main.py

 |> python Desktop¥main.py

 |% python3 Desktop/main.py

実行結果の例

Pythonista

```
  *
 * *
* * *
* * * *
* * * * *
```

This is a module from a package.

後に読み込んだパッケージにあるモジュールの命令が実行された。

```
1 from package import module
2 import module
3
4 print(module.author)
5 module.triangle('*', 5)
```

main.py

 |> python Desktop¥main.py

 |% python3 Desktop/main.py

実行結果の例

Pythonista

```
  *
 * *
* * *
* * * *
* * * * *
```

This is a module.

後にただのモジュールを読み込めばそれが実行される。

同じモジュール名が存在する場合：別名の付与

解決策

- **as文でユニークな別名**をつければ良い。

```
1 import module as m1
2 from package import module as m2
3
4 m1.triangle('*', 5)
5 m2.triangle('*', 5)
```

main.py

実行結果の例

```
      *
     * *
    * * *
   * * * *
  * * * * *
This is a module.
      *
     * *
    * * *
   * * * *
  * * * * *
This is a module from a package.
```

他のモジュール利用法：関数のみのインポートとその別名付与

```
1 from module import triangle
2
3 triangle('*', 5)
```

from文が必須。

main.py

 `> python Desktop¥main.py`

 `% python3 Desktop/main.py`

```
1 from module import triangle as t
2
3 t('*', 5)
```

main.py

 `> python Desktop¥main.py`

 `% python3 Desktop/main.py`

実行結果の例

```
  *
 * *
* * *
* * * *
* * * * *
This is a module.
```

実行結果の例

```
  *
 * *
* * *
* * * *
* * * * *
This is a module.
```

他のモジュール利用法：サブパッケージからのインポート



```
> mkdir Desktop¥package¥subpackage
> copy Desktop¥module.py Desktop¥package¥subpackage
1 個のファイルをコピーしました。
> dir Desktop¥package¥subpackage
(略)
2023/11/16  18:42                284 module.py
(略)
```



```
% mkdir Desktop/package/subpackage
% cp Desktop/module.py Desktop/package/subpackage
% ls Desktop/package
__pycache__      subpackage      module.py
% ls Desktop/package/subpackage
module.py
```

編集する

```
1  author = "Pythonista"
2  def triangle(symbol, num):
3      """moduleing a triangle with given symbol. """
4      for i in range(num):
5          for j in range((num - i) - 1):
6              print(end=" ")
7          for j in range((i + 1)):
8              print(symbol, end=" ")
9          print()
10     print("This is a module from a subpackage.")
```

編集部分。

package/subpackage/module.py

編集する

```
1  import package.subpackage.module
2                                     ディレクトリとファイルの区切りを`.`で表す。
3  package.subpackage.module.triangle('*', 5)
                                                                    main.py
```

```
1  from package.subpackage import module
2                                     aliasを付けなくても記述量が減る。
3  module.triangle('*', 5)
                                                                    main.py
```

```
1  from package.subpackage.module import triangle
2                                     メソッドのみを読み込むことができる。
3  triangle('*', 5)
                                                                    main.py
```

注意「from package.subpackage import module.triangle as m1」
のような記述はできない。from文とimport文を併用するときは、最後の要素だけ
import文で指定し、それ以外はfrom文で指定する。



```
> python Desktop¥main.py
```



```
% python3 Desktop/main.py
```

実行結果の例

```
      *
     * *
    * * *
   * * * *
  * * * * *
```

This is a module from a subpackage.

モジュールの参照場所（Python標準のsysモジュールを利用）

```
1 >>> import sys
2 >>> for place in sys.path:
3 ...     print(place)
4 ...
```



空行は実行したPythonプログラムがあるカレントディレクトリの意味。
上から下にPythonは指定されたモジュールを探しに行く。

```
C:¥Users¥trgt¥AppData¥Local¥Programs¥Python¥Python312¥python312.zip
C:¥Users¥trgt¥AppData¥Local¥Programs¥Python¥Python312¥DLLs
C:¥Users¥trgt¥AppData¥Local¥Programs¥Python¥Python312¥Lib
C:¥Users¥trgt¥AppData¥Local¥Programs¥Python¥Python312
C:¥Users¥trgt¥AppData¥Local¥Programs¥Python¥Python312¥Lib¥site-packages
```



空行は実行したPythonプログラムがあるカレントディレクトリの意味。
上から下にPythonは指定されたモジュールを探しに行く。

```
/Library/Frameworks/Python.framework/Versions/3.12/lib/python312.zip
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/lib-dynload
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages
```

timeモジュール

Pythonの time モジュールは、時間に関連する機能を提供する標準モジュール。
timeモジュールは、時間の計測、スリープ、またはフォーマットされた日付や時刻の表示など、さまざまな時間操作をサポートする。

```
1 import time
2
3 current_time = time.localtime()
4 print("Current Time:", current_time)
5
```

get-time.py

実行結果の例

```
Current Time:
time.struct_time(tm_year=2023,
tm_mon=11, tm_mday=16, tm_hour=21,
tm_min=42, tm_sec=40, tm_wday=3,
tm_yday=320, tm_isdst=0)
```

```
1 import time
2
3 print("Start")
4 time.sleep(2)
5 print("End after 2 seconds")
```

sleep.py

実行結果の例

```
Start
End after 2 seconds
```

timeモジュールの活用例：アスキーアート（Python標準のosモジュール併用）

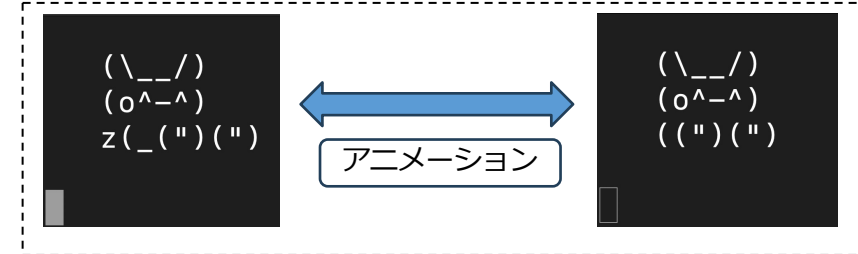
```
1 import time
2 import os
3
4 # ASCII art
5 ascii_art = [
6     r"""
7     (¥__/)
8     (o^_^)
9     z(_(")(")
10    """
11    ,
12    r"""
13    (¥__/)
14    (o^_^)
15    ((")(")
16    """
17 ]
18
19 def animate(frames, delay=0.5):
20     for frame in frames:
21         os.system('cls' if os.name == 'nt' else 'clear')
22         print(frame)
23         time.sleep(delay)
24
25 try:
26     while True:
27         animate(ascii_art)
```

raw stringと呼び、Pythonの文字列を作成する際にエスケープ文字を無視するための接頭辞。

パワーポイントの入力規則の関係で¥となっているが実際には「\（バックスラッシュ）」を意味している。

例外処理をあえて用いて、`Ctrl - C` コマンドでプログラムが綺麗に終了するようにしている。

実行結果の例



この行は、コマンドラインの画面をクリアするためのコマンド。
os.name が 'nt' ならば（Windowsの場合）、'cls' として指定されたコマンドを実行し、それ以外の場合は（Unixベースのシステムの場合）、'clear' として指定されたコマンドを実行する。その場合分けを三項演算子を用いて記述した例。

'cls': WindowsコマンドプロンプトやPowerShellで画面をクリアするためのコマンド。

'clear': Unix/Linuxのターミナルで画面をクリアするためのコマンド。

これにより、アスキーアートが表示される前にターミナルまたはコマンドプロンプトの画面をクリアして、アスキーアートが連続して表示されるようになっている。

mathモジュール

math モジュールは、数学的な操作を行うための標準モジュールで、様々な数学関数や定数をサポートする。このモジュールは、数値計算や三角関数、対数、指数関数などの数学的な操作を行うために使用される。

```
1 import math
2
3 print("The value of pi is:", math.pi)
```

pi.py

実行結果の例

The value of pi is: 3.141592653589793

```
1 import math
2
3 angle = math.pi / 4 # Angle of 45 degrees
4
5 sine_value = math.sin(angle)
6 cosine_value = math.cos(angle)
7
8 print(f"The sine of {angle} is:", sine_value)
9 print(f"The cosine of {angle} is:", cosine_value)
```

sin-cos.py

実行結果の例

The sine of 0.7853981633974483 is: 0.7071067811865475
The cosine of 0.7853981633974483 is: 0.7071067811865476

tkinterモジュール

tkinter モジュールは、PythonでGUI(Graphical User Interface)アプリケーションを構築するための標準モジュール。tkinter は、Tk GUIツールキットのPythonバインディングであり、クロスプラットフォームのGUIアプリケーションを開発するのに広く使用されている。

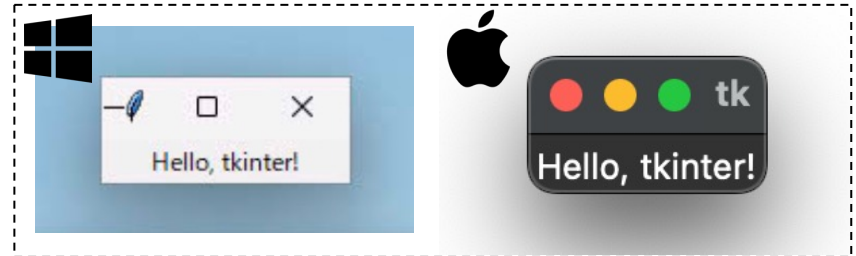
```
1 import tkinter as tk
2
3 root = tk.Tk()
4 label = tk.Label(root, text="Hello, tkinter!")
5 label.pack()
6
7 root.mainloop()
```

hello-tk.py

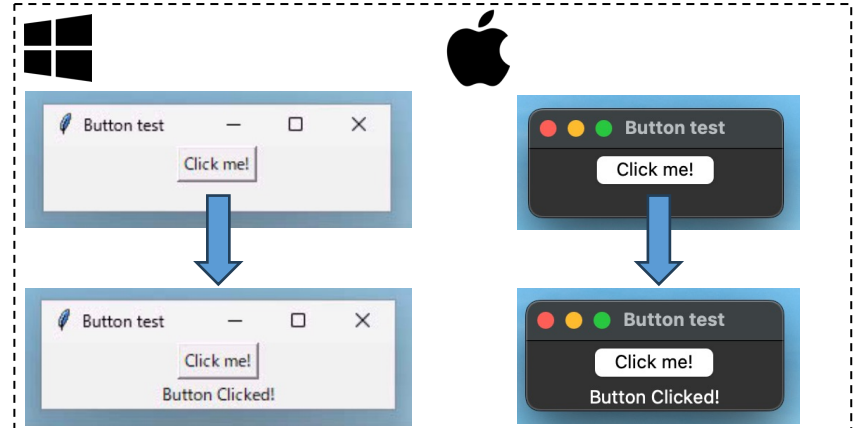
```
1 import tkinter as tk
2
3 def on_button_click():
4     label.config(text="Button Clicked!")
5
6 root = tk.Tk()
7 root.title("Button test")
8
9 button = tk.Button(root, text="Click me!", command=on_button_click)
10 button.pack()
11
12 label = tk.Label(root, text="")
13 label.pack()
14
15 root.mainloop()
```

tk-button.py

実行結果の例



実行結果の例



turtleモジュール

turtle モジュールは、Pythonでシンプルなグラフィックスを描画するための標準モジュール。turtle モジュールは、簡単なアート作品やグラフィカルなプログラムを学ぶために利用され、特に初学者や子供たちに人気がある。

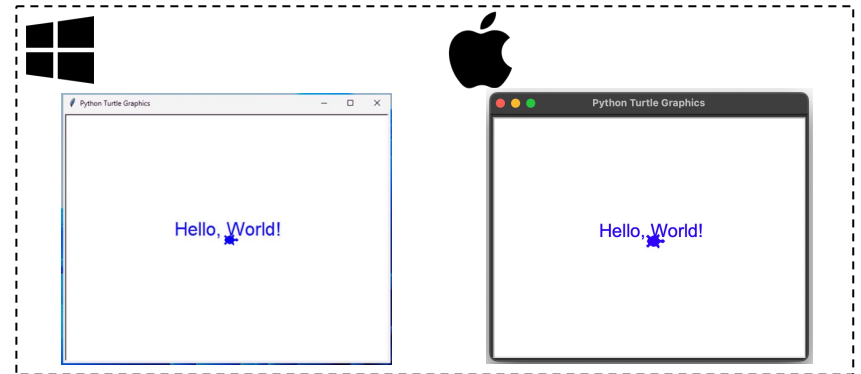
```
1 import turtle
2
3 # Window settings
4 window = turtle.Screen()
5 window.bgcolor("white")
6
7 # Turtle settings
8 t = turtle.Turtle()
9 t.shape("turtle") # Set the shape of the pen to turtle
10 t.color("blue")   # Set the pen color to blue
11
12
13
14
15 # Draw a string
16 t.write("Hello, World!", align="center", font=("Arial", 24, "normal"))
17
18 # Close the window on click
19 window.exitonclick()
```

hello-turtle.py

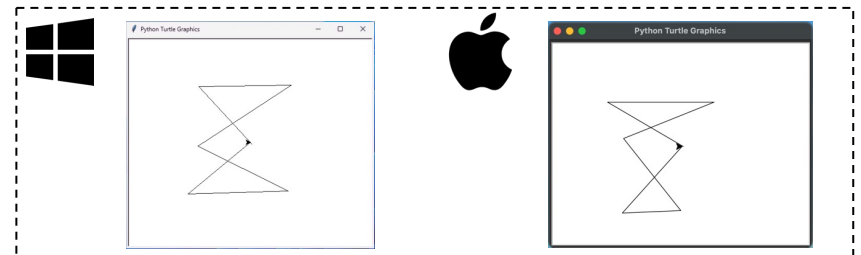
```
1 import turtle
2
3 def on_click(x, y):
4     turtle.goto(x, y)
5
6 turtle.onscreenclick(on_click)
7 turtle.done()
```

click-turtle.py

実行結果の例



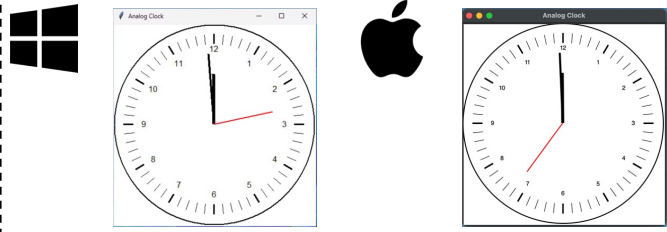
実行結果の例



標準モジュールの活用例：アナログ時計(clock.py)

```
1 import time
2 import math
3 import tkinter as tk
4
5 WIDTH = 400
6 HEIGHT = 400
7
8 root = tk.Tk()
9 root.title("Analog Clock")
10 canvas = tk.Canvas(root, width=WIDTH, height=HEIGHT,
11 bg="white")
12 canvas.pack()
13
14 def update_clock():
15     canvas.delete("all")
16     now = time.localtime()
17     hour = now.tm_hour % 12
18     minute = now.tm_min
19     second = now.tm_sec
20
21 # Draw clock face
22 canvas.create_oval(2, 2, WIDTH, HEIGHT, outline="black",
23 width=2)
24
25 # Draw hour numbers
26 for i in range(12):
27     angle = i * math.pi/6 - math.pi/2
28     x = WIDTH/2 + 0.7 * WIDTH/2 * math.cos(angle)
29     y = HEIGHT/2 + 0.7 * HEIGHT/2 * math.sin(angle)
30     if i == 0:
31         canvas.create_text(x, y, text=str(i+12), font=
32 ("Helvetica", 12), fill='black')
33     else:
34         canvas.create_text(x, y, text=str(i),
35 font=("Helvetica", 12), fill='black')
```

実行結果の例



24時間表記を12時間表記に変換。

楕円を描くメソッドを利用。第1引数と第2引数を左上、第3引数と第4引数を右下とした四角形に接する楕円が描画される。今回は円ができる。

360度(2Pi)を12等分するので約分してPi/6。

0度を12時の方向に合わせるため、-90度(-Pi/2)して角度補正。

```
36 # Draw minute lines
37 for i in range(60):
38     angle = i * math.pi/30 - math.pi/2
39     x1 = WIDTH/2 + 0.8 * WIDTH/2 * math.cos(angle)
40     y1 = HEIGHT/2 + 0.8 * HEIGHT/2 * math.sin(angle)
41     x2 = WIDTH/2 + 0.9 * WIDTH/2 * math.cos(angle)
42     y2 = HEIGHT/2 + 0.9 * HEIGHT/2 * math.sin(angle)
43     if i % 5 == 0:
44         canvas.create_line(x1, y1, x2, y2, fill="black", width=3)
45     else:
46         canvas.create_line(x1, y1, x2, y2, fill="black", width=1)
47
48 # Draw hour hand
49 hour_angle = (hour + minute/60) * math.pi/6 - math.pi/2
50 hour_x = WIDTH/2 + 0.5 * WIDTH/2 * math.cos(hour_angle)
51 hour_y = HEIGHT/2 + 0.5 * HEIGHT/2 * math.sin(hour_angle)
52 canvas.create_line(WIDTH/2, HEIGHT/2, hour_x, hour_y,
53 fill="black", width=6)
54
55 # Draw minute hand
56 minute_angle = (minute + second/60) * math.pi/30 - math.pi/2
57 minute_x = WIDTH/2 + 0.7 * WIDTH/2 * math.cos(minute_angle)
58 minute_y = HEIGHT/2 + 0.7 * HEIGHT/2 * math.sin(minute_angle)
59 canvas.create_line(WIDTH/2, HEIGHT/2, minute_x, minute_y,
60 fill="black", width=4)
61
62 # Draw second hand
63 second_angle = second * math.pi/30 - math.pi/2
64 second_x = WIDTH/2 + 0.6 * WIDTH/2 * math.cos(second_angle)
65 second_y = HEIGHT/2 + 0.6 * HEIGHT/2 * math.sin(second_angle)
66 canvas.create_line(WIDTH/2, HEIGHT/2, second_x, second_y,
67 fill="red", width=2)
68
69 canvas.after(1000, update_clock)
70
71 update_clock()
72 root.mainloop()
```

360度(2Pi)を60等分するので約分してPi/30。

360度(2Pi)を60等分するので約分してPi/30。

まとめ

- プログラムは部品化することで、**再利用性、メンテナンスの容易性、可読性、グループ開発の容易性、テストの容易性、名前空間整理の容易性**が向上する。
- 部品として見たプログラムのことを**モジュール**と呼ぶ。
- モジュールの実体は、Pythonプログラムの**ファイル**である。
- モジュールをまとめて入れた**ディレクトリ**のことを**パッケージ**と呼ぶ。
- あるPythonプログラムからモジュールを読み込むには**import文**を書く。
- **from文**をimport文と併用することで、命令の記述量を減らすことや、メソッドの読み込みができるようになる。
- モジュールやメソッドを読み込む際に**as文**でalias(別名)を付与することにより、それ以降はaliasを用いて命令を記述することができる。
- Pythonの標準モジュールとして、**sys**モジュール、**time**モジュール、**os**モジュール、**math**モジュール、**tkinter**モジュール、**turtle**モジュールを紹介。他にもあるので適宜調べること。