Створення програм з графічним інтерфейсом користувача (GUI)

У Python ви можете використовувати модуль tkinter для створення простих програм з графічним інтерфейсом.

Python не має функцій програмування графічного інтерфейсу, вбудованих у саму мову. Однак він поставляється з модулем під назвою tkinter, який дозволяє створювати прості програми з графічним інтерфейсом. Назва "tkinter" скорочено від "Tk interface". Це названо так, тому що це дає можливість програмістам Python використовувати бібліотеку графічного інтерфейсу під назвою Tk. Багато інших мов програмування також використовують бібліотеку Tk.

ПРИМІТКА. Для Python існує безліч бібліотек графічного інтерфейсу. Оскільки модуль tkinter поставляється з Python, ми будемо його використовувати.

Програма графічного інтерфейсу представляє вікно з різними графічними віджетами, з якими користувач може взаємодіяти або переглядати. Модуль tkinter містить 15 віджетів, описаних у Таблиці 13. Ми не розглянемо всі віджети tkinter у цьому розділі, але ми покажемо, як створювати прості програми з графічним інтерфейсом, які збирають вхідні дані та відображають ці дані.

Таблиця 13.

Widget	**Опис**
Button	Кнопка, яка може викликати дію при її натисканні.
Canvas	Прямокутна область, яку можна використовувати для відображення графіки.
Checkbutton	Кнопка, яка може бути в положенні "on" або "off".
Entry	Область, у якій користувач може ввести один рядок введення з клавіатури.
Frame	Контейнер, який може містити інші віджети.
Label	Область, яка відображає один рядок тексту або зображення.
Listbox	Список, з якого користувач може вибрати елемент.
Menu	Список варіантів меню, які відображаються, коли користувач натискає на віджет Menubutton.
Menubutton	Меню, яке відображається на екрані та може бути натиснене користувачем.
Message	Відображає кілька рядків тексту.
Radiobutton	Віджет, який можна вибрати або скасувати. Віджети радіокнопок зазвичай з'являються в групах і дозволяють користувачеві вибрати один з кількох варіантів.
Scale	Віджет, що дозволяє користувачеві вибрати значення, перемістивши повзунок уздовж доріжки.
Scrollbar	Може використовуватися з деякими іншими типами віджетів для забезпечення можливості прокрутки.
Text	Віджет, що дозволяє користувачеві вводити кілька рядків введення тексту.
Toplevel	Контейнер, як і фрейм, але відображається у власному вікні.

Найпростіша програма з графічним інтерфейсом, яку ми можемо продемонструвати, - це програма, яка відображає порожнє вікно. Лістинг 10 показує, як ми можемо це зробити за допомогою модуля tkinter. Під час запуску програми відображається вікно, зображене на Рис. 4. Щоб вийти з програми, просто натисніть стандартну кнопку закриття Windows (×) у верхньому правому куті вікна.

Лістинг 10. empty_window1.py

```
# This program displays an empty window.

import tkinter

def main():
    # Create the main window widget.
    main_window = tkinter.Tk()

   # Enter the tkinter main loop.
    tkinter.mainloop()

    # Call the main function.
if __name__ == '__main__':
    main()
```

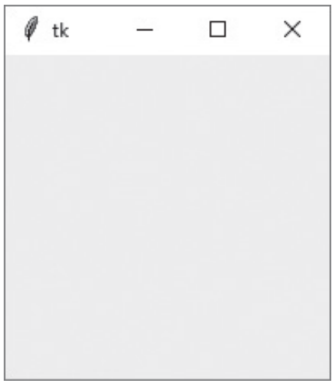

Рис. 4 Програма відображає вікно

Більшість програмістів вважають за краще використовувати об'єктно-орієнтований підхід під час написання програми з графічним інтерфейсом. Замість того, щоб писати функцію для створення екранних елементів програми, звичайною практикою є написання класу методом __init__, який створює графічний інтерфейс. Коли створюється екземпляр класу, на екрані з'являється графічний інтерфейс. Для демонстрації Лістинг 11 показує об’єктно-орієнтовану версію нашої програми, яка відображає порожнє вікно. Коли ця програма запускається, вона відображає вікно, зображене на Рис. 4.

Лістинг 11. Об’єктно-орієнтована версія empty_window2.py

```
# This program displays an empty window.

import tkinter

class MyGUI:
    def __init__(self):
        # Create the main window widget.
        self.main_window = tkinter.Tk()
```

```
        # Enter the tkinter main loop.
        tkinter.mainloop()

# Create an instance of the MyGUI class.
if __name__ == '__main__':
    my_gui = MyGUI()
```

Наступна діаграма показує три компоненти програмування графічного інтерфейсу:

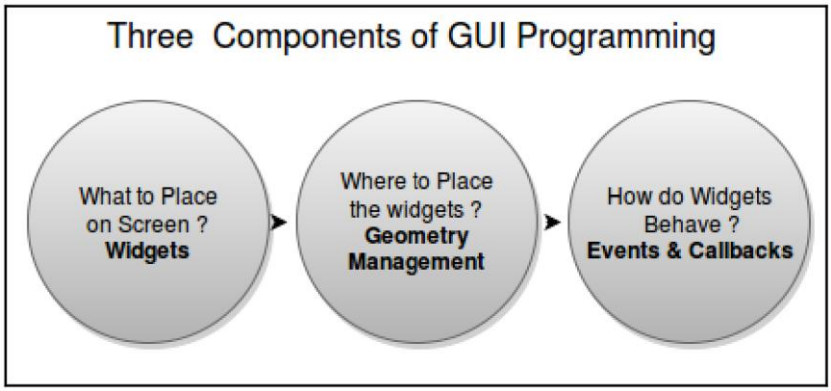

Рис. 5. Три аспекти GUI

Перш за все треба вирішити які компоненти (виджети) повинні з'являтися на екрані.

Потім необхідно розмістити віджети на екрані. Це включає вирішення положення та структурного планування різних компонентів. У Tkinter це називається управлінням геометрією (geometry management).

Після цього треба вирішити як компоненти взаємодіють і поводяться. Це передбачає додавання функціональності кожному компоненту. Кожен компонент або віджет щось робить. Наприклад, при натисканні на кнопку щось відбувається у відповідь. Смуга прокручування обробляє прокрутку, а прапорці та перемикачі дозволяють користувачам зробити певний вибір. У Tkinter функціональні можливості різних віджетів управляються прив'язкою команд або прив'язкою подій за допомогою зворотного виклику.

Відображення тексту за допомогою віджетів Label

Ви можете використовувати віджет Label для відображення одного рядка тексту у вікні. Щоб створити віджет Label, потрібно створити екземпляр класу Label модуля tkinter. Програма на Лістингу 12 створює вікно, що містить віджет Label, який відображає текст “Hello World!”. Вікно показано на Рис. 6.

Лістинг 12. hello_world.py

```
import tkinter

class MyGUI:
    def __init__(self):
    # Create the main window widget.
        self.main_window = tkinter.Tk()
        self.label = tkinter.Label(self.main_window,
                                   text='Hello World!')
```

```
        # Call the Label widget's pack method.
        self.label.pack()

        # Enter the tkinter main loop.
        tkinter.mainloop()

# Create an instance of the MyGUI class.
if __name__ == '__main__':
    my_gui = MyGUI()
```

Рис. 6 Вікно програми

Метод pack визначає, де має розміщуватися віджет, і робить його видимим під час відображення головного вікна. (Ви викликаєте метод pack для кожного віджета у вікні.)

Організація віджетів за допомогою Frames

Frame - це контейнер, який може містити інші віджети. Ви можете використовувати Frames для організації віджетів у вікні.

Наприклад, ви можете розмістити набір віджетів в одному фреймі та упорядкувати їх певним чином, а потім розмістити набір віджетів в іншому фреймі та розташувати їх по-іншому. Програма на Лістингу 13 це демонструє. Коли програма запускається, відображається вікно, зображене на Рис. 7.

Лістинг 13. frame_demo.py

```
import tkinter

class MyGUI:
    def __init__(self):
        self.main_window = tkinter.Tk()
        self.top_frame = tkinter.Frame(self.main_window)
        self.bottom_frame = tkinter.Frame(self.main_window)

        self.label1 = tkinter.Label(self.top_frame,
                                    text='Winken')
        self.label2 = tkinter.Label(self.top_frame,
                                    text='Blinken')
        self.label3 = tkinter.Label(self.top_frame,
                                    text='Nod')
        self.label1.pack(side='top')
        self.label2.pack(side='top')
        self.label3.pack(side='top')

        self.label4 = tkinter.Label(self.bottom_frame,
                                    text='Winken')
        self.label5 = tkinter.Label(self.bottom_frame,
                                    text='Blinken')
        self.label6 = tkinter.Label(self.bottom_frame,
                                    text='Nod')
        self.label4.pack(side='left')
```

```
        self.label5.pack(side='left')
        self.label6.pack(side='left')

        self.top_frame.pack()
        self.bottom_frame.pack()

        tkinter.mainloop()

if __name__ == '__main__':
    my_gui = MyGUI()
```

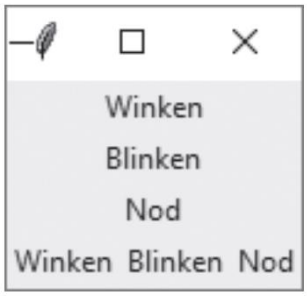

Рис. 7 Два фрейми

Віджети та діалогові вікна інформації

КОНЦЕПЦІЯ: Ви використовуєте віджет Button для створення стандартної кнопки у вікні. Коли користувач натискає кнопку, викликається певна функція або метод.

Інформаційне діалогове вікно - це просте вікно, яке відображає повідомлення користувачеві, і має кнопку OK, яка закриває діалогове вікно. Ви можете використовувати функцію showinfo модуля tkinter.messagebox для відображення діалогового вікна інформації.

Кнопка - це віджет, який користувач може натиснути, щоб викликати дію. Під час створення віджета кнопки можна вказати текст, який має відображатися на лицьовій стороні кнопки, та назву функції зворотного виклику. Функція зворотного виклику - це функція або метод, який виконується, коли користувач натискає кнопку.

ПРИМІТКА. Функція зворотного виклику також відома як обробник подій (event handler), оскільки вона обробляє подію, яка відбувається, коли користувач натискає кнопку.

Для демонстрації ми розглянемо програму на Лістингу 14. Ця програма відображає вікно, зображене на Рис. 8. Коли користувач натискає кнопку, програма відображає окреме інформаційне діалогове вікно, показане на Рис. 9. Для відображення діалогового вікна інформації ми використовуємо функцію showinfo, яка знаходиться в модулі tkinter.messagebox. (Щоб використовувати функцію showinfo, вам потрібно імпортувати модуль tkinter.messagebox.) Це загальний формат виклику функції showinfo:

```
tkinter.messagebox.showinfo(title, message)
```

У загальному форматі `title` - це рядок, який відображається у рядку заголовка діалогового вікна, а `message` - інформаційний рядок, який відображається у головній частині діалогового вікна.

Лістинг 14. button_demo.py

```
import tkinter
import tkinter.messagebox

class MyGUI:
    def __init__(self):
        self.main_window = tkinter.Tk()
        self.my_button = tkinter.Button(self.main_window,
             text='Click Me!', command=self.do_something)
        self.my_button.pack()
        tkinter.mainloop()

    def do_something(self):
        tkinter.messagebox.showinfo('Response',
                    'Thanks for clicking the button.')

if __name__ == '__main__':
    my_gui = MyGUI()
```

Рис. 8 Вікно програми

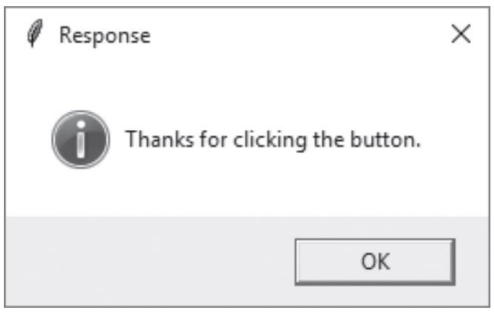

Рис. 9 Інформаційне діалогове вікно

Отримання вводу за допомогою віджета Entry

КОНЦЕПЦІЯ: Віджет Entry - це прямокутна область, у яку користувач може вводити дані. Ви використовуєте метод get віджета Entry для отримання даних, які були набрані у віджеті.

Віджет Entry - це прямокутна область, в яку користувач може вводити текст. Віджети входу використовуються для збору вхідних даних у програмі графічного інтерфейсу. Як правило, програма матиме один або кілька віджетів Entry у вікні разом із кнопкою, яку користувач натискає, щоб надіслати дані, які він ввів у віджети Entry. Функція зворотного виклику кнопки витягує дані з віджетів входу вікна та обробляє їх.

Ви використовуєте метод get віджета Entry для отримання даних, які користувач ввів у віджет. Метод get повертає рядок, тому його доведеться конвертувати у відповідний тип даних, якщо віджет Entry використовується для числового введення.

Для демонстрації ми розглянемо програму, яка дозволяє користувачеві ввести відстань у кілометрах у віджет Entry, а потім натиснути кнопку, щоб

побачити цю відстань, перетворену в милі. Формула перетворення кілометрів у милі така:

```
Miles = Kilometers * 0.6214
```

На Рис. 10 показано вікно, яке відображає програма. Щоб розташувати віджети у положеннях, показаних на малюнку, ми організуємо їх у два фрейми, як показано на Рис. 11. Мітка, яка відображає запит та віджет Entry, зберігатиметься у top_frame, а їх методи pack будуть викликані з аргументом side = 'left'. Це призведе до їх горизонтального відображення в фреймі. Кнопки Convert та Quit зберігатимуться у bottom_frame, а також їх методи pack також будуть викликатися з аргументом side = 'left'.

Програма на Лістингу 15 показує код програми. На Рис. 12 показано, що відбувається, коли користувач вводить 1000 у віджет Entry і натискає кнопку Convert.

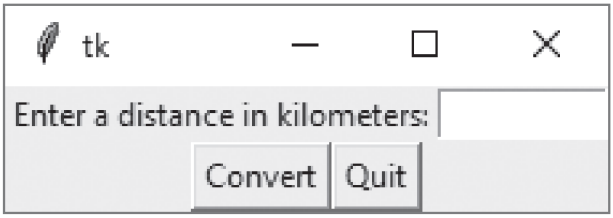

Рис. 10 Вікно програми

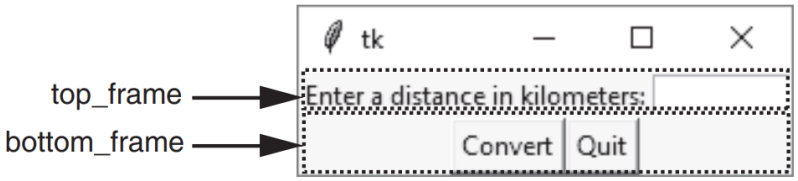

Рис. 11. Розташування віджетів

Лістинг 15. kilo_converter.py

```
import tkinter
import tkinter.messagebox

class KiloConverterGUI:
    def __init__(self):
        self.main_window = tkinter.Tk()
        self.top_frame = tkinter.Frame(self.main_window)
        self.bottom_frame = tkinter.Frame(self.main_window)

        self.prompt_label = tkinter.Label(self.top_frame,
                    text='Enter a distance in kilometers:')
        self.kilo_entry = tkinter.Entry(self.top_frame,
                                        width=10)
        self.prompt_label.pack(side='left')
        self.kilo_entry.pack(side='left')

        self.calc_button = tkinter.Button(self.bottom_frame,
                     text='Convert', command=self.convert)
        self.quit_button = tkinter.Button(self.bottom_frame
            , text='Quit', command=self.main_window.destroy)
        self.calc_button.pack(side='left')
        self.quit_button.pack(side='left')
```

```
            self.top_frame.pack()
            self.bottom_frame.pack()

            tkinter.mainloop()

        def convert(self):
            kilo = float(self.kilo_entry.get())
            miles = kilo * 0.6214
            tkinter.messagebox.showinfo('Results', str(kilo) +
             ' kilometers is equal to ' + str(miles) + ' miles.')

if __name__ == '__main__':
    kilo_conv = KiloConverterGUI()
```

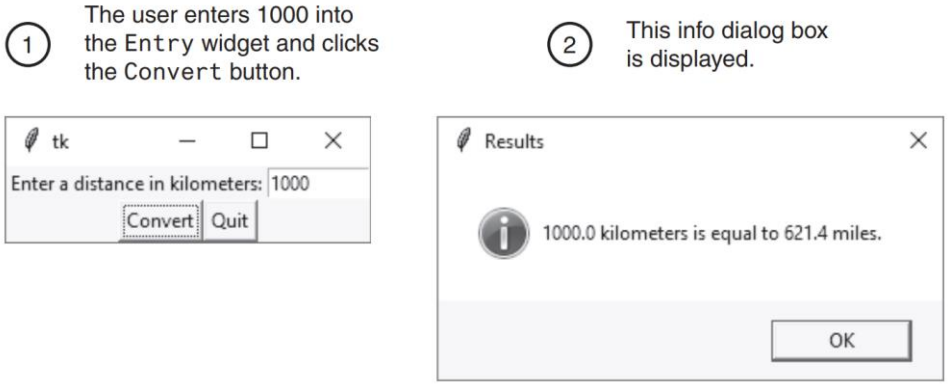

Рис. 12 Результат в інформаційній панелі

Використання Labels як полів виводу

КОНЦЕПЦІЯ: Коли об’єкт StringVar асоціюється з віджетом Label, віджет Label відображає будь-які дані, що зберігаються в об’єкті StringVar.

Раніше ви бачили, як використовувати інформаційне діалогове вікно для відображення результатів. Якщо ви не хочете відображати окреме діалогове вікно для виведення програми, ви можете використовувати віджети Label у головному вікні програми для динамічного відображення результатів. Ви просто створюєте порожні віджети Label у своєму головному вікні, а потім вводите код, який відображає потрібні дані у цих мітках при натисканні кнопки.

Модуль tkinter надає клас з назвою StringVar, який можна використовувати разом з віджетом Label для відображення даних. По-перше, ви створюєте об'єкт StringVar. Потім ви створюєте віджет Label і пов'язуєте його з об'єктом StringVar. З цього моменту будь-яке значення, яке потім зберігається в об’єкті StringVar, автоматично відображатиметься у віджеті Label.

Програма на Лістингу 16 демонструє, як це зробити. Це модифікована версія програми kilo_converter, яку ви бачили на Лістигу 15. Замість того, щоб спливати інформаційне діалогове вікно, ця версія програми відображає кількість миль на мітці у головному вікні.

Лістинг 16. kilo_converter2.py

```
import tkinter

class KiloConverterGUI:
```

```
    def __init__(self):
        self.main_window = tkinter.Tk()
        self.top_frame = tkinter.Frame()
        self.mid_frame = tkinter.Frame()
        self.bottom_frame = tkinter.Frame()

        self.prompt_label = tkinter.Label(self.top_frame,
                  text='Enter a distance in kilometers:')
        self.kilo_entry = tkinter.Entry(self.top_frame,
                                        width=10)
        self.prompt_label.pack(side='left')
        self.kilo_entry.pack(side='left')

        self.descr_label = tkinter.Label(self.mid_frame,
                               text='Converted to miles:')
        self.value = tkinter.StringVar()
        self.miles_label = tkinter.Label(self.mid_frame,
                                  textvariable=self.value)
        self.descr_label.pack(side='left')
        self.miles_label.pack(side='left')

        self.calc_button = tkinter.Button(self.bottom_frame,
                        text='Convert', command=self.convert)
        self.quit_button = tkinter.Button(self.bottom_frame,
              text='Quit', command=self.main_window.destroy)
        self.calc_button.pack(side='left')
        self.quit_button.pack(side='left')

        self.top_frame.pack()
        self.mid_frame.pack()
        self.bottom_frame.pack()

        tkinter.mainloop()

    def convert(self):
        kilo = float(self.kilo_entry.get())
        miles = kilo * 0.6214
        self.value.set(miles)

if __name__ == '__main__':
    kilo_conv = KiloConverterGUI()
```

Коли ця програма працює, вона відображає вікно, зображене на Рис. 13. На Рис. 14 показано, що відбувається, коли користувач вводить 1000 за кілометри і натискає кнопку Convert. Кількість миль відображається на мітці у головному вікні.

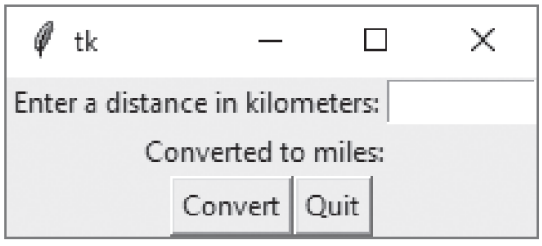

Рис. 13 Вікно модифікованої програми

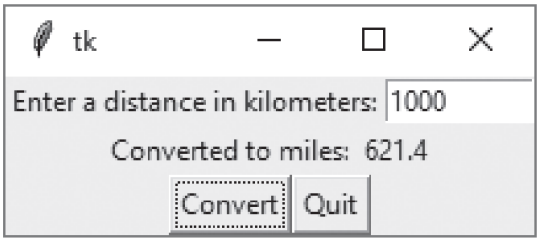

Рис. 14 Після введення 1000 і натискання кнопки Convert

Бібліотека Matplotlib

Matplotlib - це найпростіша бібліотека для графічної візуалізації даних у Python. Вона включає майже всі види графіків, які ви можете собі уявити. Те, що вона базова, не означає, що вона не є потужною, багато інших бібліотек візуалізації даних, про які ми будемо говорити, базуються на ній.

Matplotlib - родоначальник бібліотек візуалізації даних Python. Незважаючи на те, що їй більше десяти років, вона все ще є найбільш широко використовуваною бібліотекою для створення графіків у спільноті Python. Вона була розроблена, щоб дуже нагадувати MATLAB, фірмову мову програмування, розроблену у 1980-х роках.

Оскільки Matplotlib була першою бібліотекою візуалізації даних Python, багато інших бібліотек побудовані на її основі або призначені для роботи в парі з нею під час аналізу. Деякі бібліотеки, такі як pandas та Seaborn, є "обгортками" над Matplotlib. Вони дозволяють отримати доступ до кількох методів Matplotlib з меншим кодом.

Matplotlib спочатку був написаний Джоном Д. Хантером (John D. Hunter), з тих пір він має активну спільноту розробників і розповсюджується за ліцензією BSD.

Matplotlib - це бібліотека для створення 2D графіків масивів у Python. Хоча вона походить від імітації графічних команд MATLAB, вона не залежить від MATLAB і може використовуватися в Pythonic, об'єктно-орієнтованому стилі. Хоча Matplotlib написаний переважно на чистому Python, він широко використовує NumPy та інші коди розширень, щоб забезпечити хорошу продуктивність навіть для великих масивів. Matplotlib розроблений з філософією того, що ви повинні мати можливість створювати прості сюжети за допомогою кількох команд або лише однієї! Якщо ви хочете побачити гістограму своїх даних, вам не потрібно створювати екземпляри об’єктів, викликати методи, задавати властивості тощо; це має просто працювати.

Код Matplotlib концептуально поділений на три частини:

- Інтерфейс pylab - це набір функцій, наданих matplotlib.pylab, що дозволяє користувачеві створювати графіки з кодом, дуже подібним до коду, що генерує фігури MATLAB.
- Інтерфейс Matplotlib або API Matplotlib - це набір класів, які виконують важку роботу, створюють та керують фігурами, текстом, рядками, графіками, тощо. Це абстрактний інтерфейс, який нічого не знає про виведення.

- Бекенди - це пристрої для малювання, залежні від пристрою, також відомі як візуалізатори, які виводять frontend зображення на друк або на пристрій відображення.

Як користуватися Matplotlib?

Matplotlib зображує ваші дані на фігурах (Figures), кожна з яких може містити одну або кілька осей (Axes, тобто областей, де точки можна вказати з точки зору координат x-y (або тета-r у полярній сюжет, або x-y-z у тривимірному графіку, тощо). Найпростіший спосіб створення фігури з осями - це використання pyplot.subplots. Потім ми можемо використовувати Axes.plot для нанесення деяких даних по осях:

```
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots() #Create a figure containing a single axes
```

Наведений вище рядок коду створює два об’єкти і призначає їх змінним "fig" та "ax" відповідно. Спробуємо викликати перший об’єкт:

```
fig
plt.show()
```

На виході отримаємо:

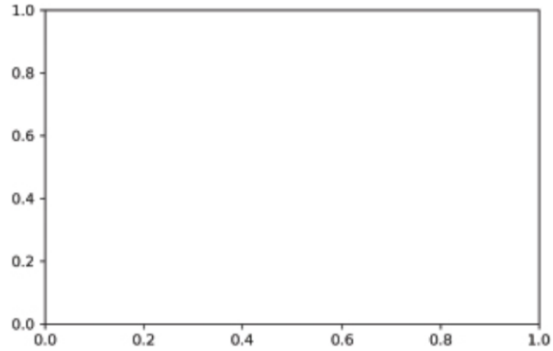

Рис. 15 Фігура Matplotlib, що містить тільки осі

Як бачите, це просто чисте полотно, на якому ви можете намалювати свій графік. Тож почнемо малювати. Інший об’єкт - це об’єкт осей, на якому будуть побудовані ділянки.

```
fig
ax.plot([1, 2, 3, 4], [1, 4, 2, 3])
plt.show()
```

Результат буде таким:

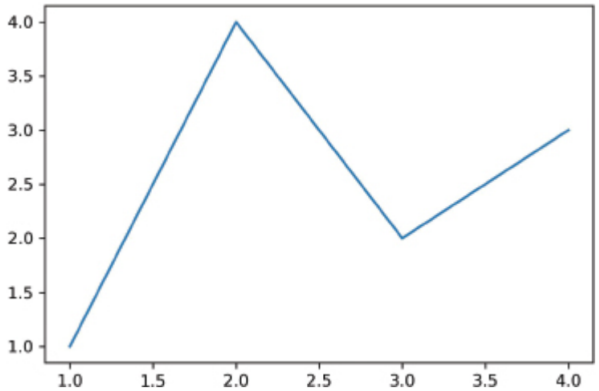

Рис. 16 Малювання ліній на об’єкті осей

Багато інших бібліотек або мов побудови графіків не вимагають явного створення осей. Ви можете зробити те ж саме в Matplotlib: для кожного методу осей для графічного відображення існує відповідна функція в модулі matplotlib.pyplot, яка виконує цей графік на "поточних" осях, створюючи ці осі (і їх батьківську фігуру), якщо вони ще не існують. Тому попередній приклад можна записати коротше як

```
plt.plot([1, 2, 3, 4], [1, 4, 2, 3])
plt.show()
```

Інтерфейси Matplotlib

Існує два способи використання Matplotlib:

- Об'єктно-орієнтований інтерфейс.
- Інтерфейс pyplot.

Об'єктно-орієнтований інтерфейс

Якщо ви підете саме цим способом використання Matplotlib, то ви повинні явно створити фігури та осі та викликати на них методи ("об’єктно-орієнтований (ОО) стиль").

Приклад інтерфейсу OO-Style:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 2, 100)
fig, ax = plt.subplots()
ax.plot(x, x, label = "linear")
ax.plot(x, x**2, label = "quardratic")
ax.plot(x, x**3, label = "cubic")
ax.set_xlabel('x label')
ax.set_ylabel('y label')
ax.set_title("Simple Plot")
ax.legend()
plt.show()
```

Отримаємо графіки, показані на Рис. 17.

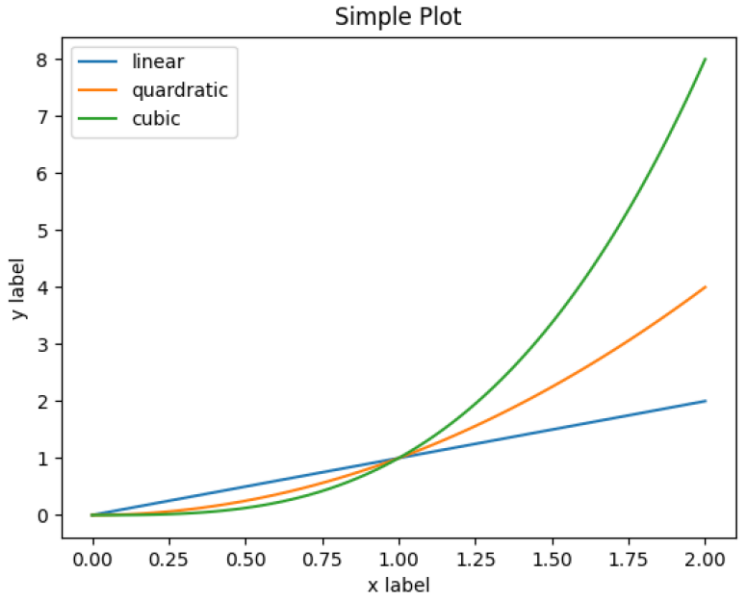

Рис. 17 Графіки функцій y=x, y=x**2 y=x**3

Інтерфейс pyplot

Якщо ви йдете саме цим способом використання Matplotlib, то замість того, щоб створювати фігури та осі вручну, ви покладаєтесь на pyplot для автоматичного створення та керування фігурами та осями та використовуєте функції pyplot для побудови графіків.

Той самий приклад, який ми бачили вище, тепер ми побачимо з інтерфейсом pyplot для порівняння.

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 2, 100)
plt.plot(x, x, label = "linear")
plt.plot(x, x**2, label = "quardratic")
plt.plot(x, x**3, label = "cubic")
plt.xlabel('x label')
plt.ylabel('y label')
plt.title("Simple Plot")
plt.legend()
plt.show()
```

Результат буде таким самим, як на Рис. 17.

Для відображення сітки на графіку можна додати рядок

```
plt.grid()
```

Основні елементи графіка Matplotlib

Основні елементи графіка Matplotlib показані на Рис. 18 (https://matplotlib.org/stable/gallery/showcase/anatomy.html).

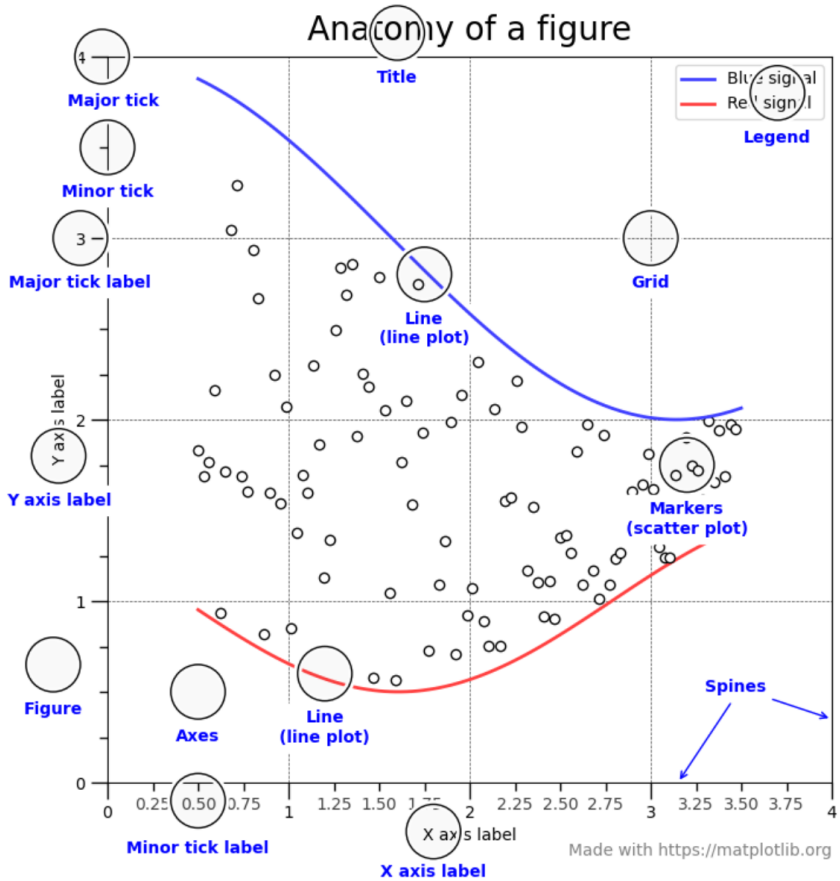

Рис. 18 Основні елементи графіка Matplotlib

Кореневим елементом при побудові графіків в системі Matplotlib є Фігура (Figure). Все, що намальовано на рисунку вище є елементами фігури. Розглянемо її складові більш докладно.

Графік

На рисунку представлені три графіка – два лінійних та точковий. Matplotlib надає величезну кількість різних налаштувань, які можна використовувати для того, щоб надати графікам необхідний вид: задати колір, товщину, тип, стиль лінії і багато іншого.

Осі

Другим, після безпосередньо самого графіка, за важливістю елементом фігури є осі. Для кожної осі можна задати мітку (підпис), основні (major) і додаткові (minor) елементи шкали, їх підписи, розмір, товщину і діапазони.

Сітка і легенда

Сітка і легенда є елементами фігури, які значно підвищують інформативність графіка. Сітка може бути основною (major) і додатковою (minor). Кожному типу сітки можна задавати колір, товщину лінії і тип. Для відображення сітки і легенди використовуються відповідні команди.

Код, за допомогою якого була побудована фігура, зображена на Рис. 18 представлений за посиланням на початку цього підрозділу.

Поверхні і лінії рівня

Matplotlib також може побудувати тривимірні дані різними способами. Два поширених варіанти для відображення тривимірних даних – це використання поверхневих або контурних графіків (як контурні лінії на карті). У цьому підрозділі ми розглянемо метод для побудови поверхонь із тривимірних даних і побудуємо контури тривимірних даних.

Щоб побудувати тривимірні дані, їх потрібно впорядкувати у двовимірні масиви для компонентів x, y та z, де обидва компоненти x та y повинні мати ту саму форму, що й компонент z. Для цієї демонстрації ми побудуємо поверхню, що відповідає функції $z=f(x, y) = x^2y^3$ в діапазоні $-2 \le x \le 2$ і $-1 \le y \le 1$.

Перше завдання полягає в тому, щоб створити відповідну сітку з пар (x, y), на якій можна оцінити цю функцію:

1. Спочатку ми використовуємо np.linspace для створення розумної кількості точок у цих діапазонах:
   ```
   X = np.linspace(-2, 2)
   Y = np.linspace(-1, 1)
   ```
2. Тепер нам потрібно створити сітку, на якій будемо обчислювати значення z. Для цього ми використовуємо метод np.meshgrid:
   ```
   x, y = np.meshgrid(X, Y)
   ```
3. Тепер ми можемо обчислити значення z в кожній з точок сітки:
   ```
   z = x**2 * y**3
   ```
4. Щоб побудувати тривимірні поверхні, нам потрібно завантажити панель інструментів Matplotlib, mplot3d, яка постачається з пакетом Matplotlib. Це не буде використовуватися явно в коді, але за

лаштунками це робить утиліти для тривимірного графіка доступними для Matplotlib:

```
from mpl_toolkits import mplot3d
```

5. Далі створюємо нову фігуру і набір тривимірних осей для фігури:

```
fig = plt.figure()
ax = fig.add_subplot(projection="3d")
```

6. Тепер ми можемо викликати метод plot_surface на цих осях, щоб побудувати дані:

```
ax.plot_surface(x, y, z)
```

7. Надзвичайно важливо додавати мітки осей до тривимірних графіків, оскільки може бути не зрозуміло, яка вісь яка на відображеній діаграмі:

```
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")
ax.set_zlabel("$z$")
```

8. На цьому етапі ми також повинні встановити назву:

```
ax.set_title("Graph of the function $f(x) = x^2y^3$)
```

Ви можете використовувати метод plt.show для відображення фігури в новому вікні або plt.savefig, щоб зберегти фігуру у файлі. Результат попередньої послідовності показано тут:

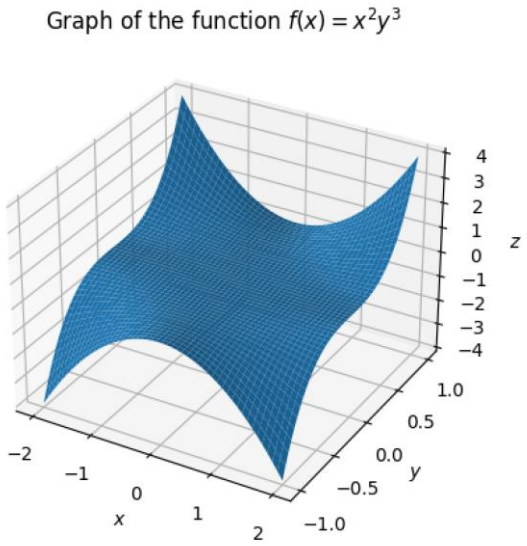

Рис. 19 Графік функції $z = x^2y^3$

9. Для контурних графіків не потрібен набір інструментів mplot3d, а в інтерфейсі pyplot є процедура contour, яка створює контурні графіки. Однак, на відміну від звичайних (двовимірних) програм побудови графіків, підпрограма contour вимагає тих самих аргументів, що й метод plot_surface. Ми використовуємо таку послідовність для створення сюжету:

```
fig = plt.figure() # Force a new figure
plt.contour(x, y, z)
plt.title("Contours of $f(x) = x^2y^3$")
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.show()
```

Результат показаний на наступному графіку:

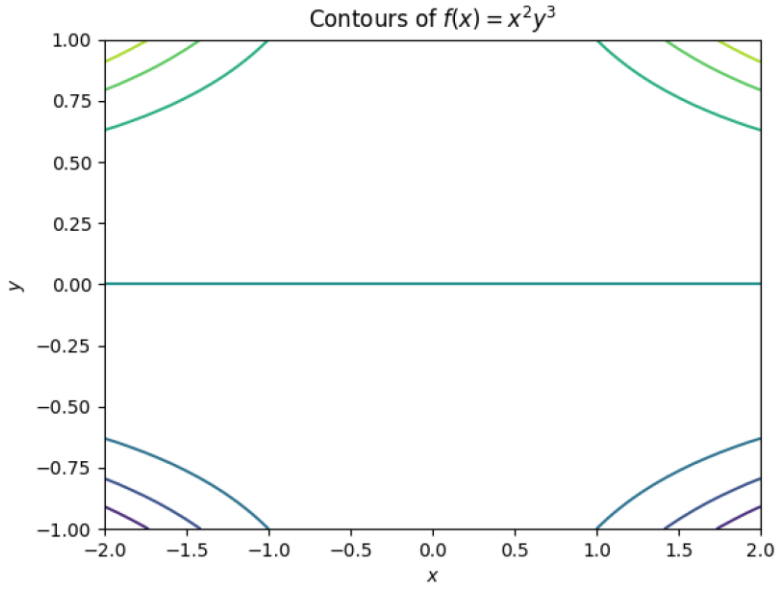

Рис. 20 Графік ліній рівня

Робота з Seaborn

Seaborn — це пакет Python для візуалізації даних, який також забезпечує високорівневий інтерфейс для Matplotlib. З Seaborn легше працювати, ніж Matplotlib, і насправді він розширює Matplotlib, але майте на увазі, що Seaborn не такий потужний, як Matplotlib.

Seaborn вирішує дві проблеми Matplotlib. Перша включає параметри Matplotlib за замовчуванням. Seaborn працює з різними параметрами, що забезпечує більшу гнучкість, ніж відтворення графіків Matplotlib за замовчуванням. Seaborn усуває обмеження значень за замовчуванням Matplotlib для таких функцій, як кольори, галочки на верхній і правій осях і стиль (серед інших).

Крім того, Seaborn полегшує побудову цілих фреймів даних (подібно до Pandas), робити це в Matplotlib важче. Тим не менш, оскільки Seaborn розширює Matplotlib, знання Matplotlib є вигідним і спростить вашу криву навчання.

Інсталяцію пакета Seaborn можна виконати командою

```
pip install seaborn
```

Можливості Seaborn

Деякі з можливостей Seaborn включають:

• масштабування графіка Seaborn
• задання стилю графіка
• встановлення розміру фігури
• обертання тексту напису
• встановлення xlim або ylim
• встановлення логарифмічного масштабу
• додавання заголовку

Деякі корисні методи:

• plt.xlabel()

• plt.ylabel()
• plt.annotate()
• plt.legend()
• plt.ylim()
• plt.savefig()

Seaborn підтримує різні вбудовані набори даних, як і NumPy і Pandas, включаючи набір даних Iris і Titanic, обидва з яких ви побачите в наступних розділах. Як відправну точку, трирядковий зразок коду в наступному розділі показує, як відобразити рядки у вбудованому наборі даних «tips».

Вбудовані набори даних Seaborn

У наступному листингі показано вміст файлу seaborn_tips.py, який ілюструє, як читати набір даних tips у фрейм даних і відображати перші п’ять рядків набору даних.

```
import seaborn as sns
df = sns.load_dataset("tips")
print(df.head())
```

Цей листинг дуже простий: після імпорту seaborn змінна df ініціалізується даними з вбудованого набору даних tips, а оператор print() відображає перші п’ять рядків df. Зверніть увагу, що API load_dataset() шукає онлайн або вбудовані набори даних. На виході отримаємо:

```
   total_bill   tip     sex smoker  day    time   size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

Набір даних Iris в Seaborn

У наступному листингі показано вміст файлу seaborn_iris.py, який ілюструє, як відобразити набір даних Iris.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Load iris data
iris = sns.load_dataset("iris")
# Construct iris plot
sns.swarmplot(x="species", y="petal_length", data=iris)
# Show plot
plt.show()
```

Програма спочатку імпортує seaborn і matplotlib.pyplot, а потім ініціалізує змінну iris вмістом вбудованого набору даних Iris. Далі API swamplot() відображає графік з горизонтальною віссю, позначеною species, і вертикальною віссю з позначкою petal_length, а відображені точки є з набору даних Iris.

На Рис. 19 показано зображення набору даних Iris на основі коду програми seaborn_iris.py.

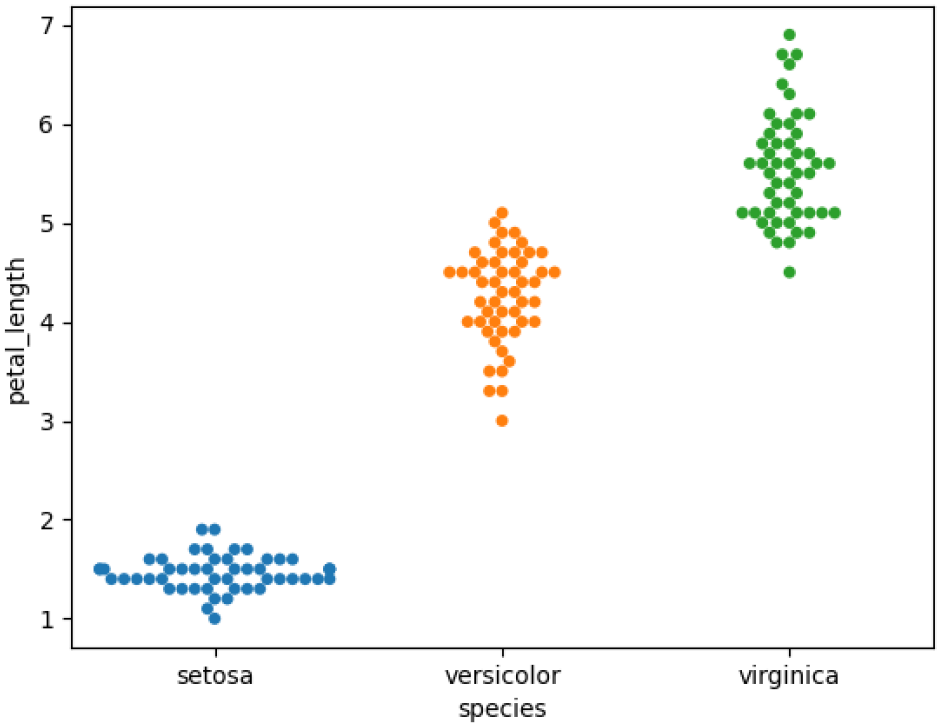

Рис. 19 Набір даних Iris

Набір даних Titanic в Seaborn

Наступний листинг відображає вміст файлу seaborn_titanic_plot.py, який ілюструє, як відобразити дані з вбудованого набіру даних Titanic.

```
import matplotlib.pyplot as plt
import seaborn as sns
titanic = sns.load_dataset("titanic")
g = sns.factorplot("class", "survived", "sex",
data=titanic, kind="bar", palette="muted", legend=False)
plt.show()
```

Листинг містить ті самі оператори імпорту, що й попередня програма, а потім ініціалізує змінну titanic вмістом вбудованого набору даних Titanic. Далі API factorplot() відображає графік з атрибутами набору даних, які вказані у виклику API.

На Рис. 20 показано графік даних з набору даних Titanic на основі коду програми.

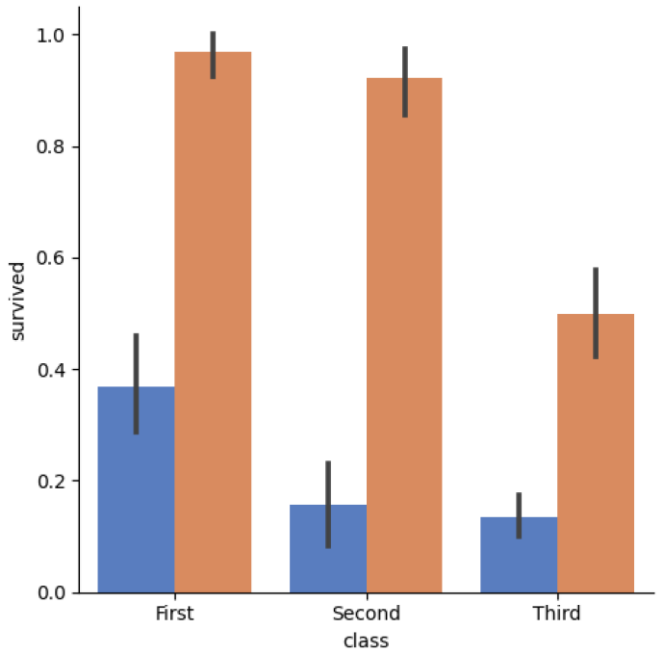

Рис. 20 Гістограма для набору даних Titanic

Витягування даних з набору даних Titanic в Seaborn

У наступному листингі показано вміст файлу seaborn_titanic.py, який ілюструє, як витягувати підмножини даних із набору даних Titanic.

```
import matplotlib.pyplot as plt
import seaborn as sns
titanic = sns.load_dataset("titanic")
print("titanic info:")
titanic.info()
print("first five rows of titanic:")
print(titanic.head())
print("first four ages:")
print(titanic.loc[0:3,'age'])
print("fifth passenger:")
print(titanic.iloc[4])
#print("first five ages:")
#print(titanic['age'].head())
#print("first five ages and gender:")
#print(titanic[['age','sex']].head())
#print("descending ages:")
#print(titanic.sort_values('age', ascending = False).head())
#print("older than 50:")
#print(titanic[titanic['age'] > 50])
#print("embarked (unique):")
#print(titanic['embarked'].unique())
#print("survivor counts:")
#print(titanic['survived'].value_counts())
#print("counts per class:")
#print(titanic['pclass'].value_counts())
#print("max/min/mean/median ages:")
#print(titanic['age'].max())
#print(titanic['age'].min())
#print(titanic['age'].mean())
```

```
#print(titanic['age'].median())
```

Листинг містить ті самі оператори імпорту, що й попередні листинги, а потім ініціалізує змінну titanic вмістом вбудованого набору даних Titanic. Наступна частина лістингу відображає різні аспекти набору даних Titanic, такі як його структура, перші п’ять рядків, перші чотири віки (ages) та деталі п’ятого пасажира.

Як бачите, існує великий блок «прокоментованого» коду, який ви можете розкоментувати, щоб побачити пов’язані результати, такі як вік, стать, особи старше 50 років, унікальні рядки тощо. Вихід програми буде таким:

```
titanic info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 15 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   survived     891 non-null    int64
 1   pclass       891 non-null    int64
 2   sex          891 non-null    object
 3   age          714 non-null    float64
 4   sibsp        891 non-null    int64
 5   parch        891 non-null    int64
 6   fare         891 non-null    float64
 7   embarked     889 non-null    object
 8   class        891 non-null    category
 9   who          891 non-null    object
 10  adult_male   891 non-null    bool
 11  deck         203 non-null    category
 12  embark_town  889 non-null    object
 13  alive        891 non-null    object
 14  alone        891 non-null    bool
dtypes: bool(2), category(2), float64(2), int64(4), object(5)
memory usage: 80.7+ KB
first five rows of titanic:
   survived  pclass     sex   age  sibsp  parch     fare embarked  class
who  adult_male deck  embark_town alive  alone
0         0       3    male  22.0      1      0   7.2500        S  Third
man        True  NaN  Southampton    no  False
1         1       1  female  38.0      1      0  71.2833        C  First
woman       False    C    Cherbourg   yes  False
2         1       3  female  26.0      0      0   7.9250        S  Third
woman       False  NaN  Southampton   yes   True
3         1       1  female  35.0      1      0  53.1000        S  First
woman       False    C  Southampton   yes  False
4         0       3    male  35.0      0      0   8.0500        S  Third
man        True  NaN  Southampton    no   True
first four ages:
0    22.0
1    38.0
2    26.0
3    35.0
Name: age, dtype: float64
fifth passenger:
survived                 0
pclass                   3
sex                   male
age                   35.0
```

```
sibsp                    0
parch                    0
fare                  8.05
embarked                 S
class                Third
who                    man
adult_male            True
deck                   NaN
embark_town    Southampton
alive                   no
alone                 True
Name: 4, dtype: object
```

Бібліотека NumPy

Вектори, матриці та багатовимірні масиви є важливими інструментами чисельних обчислень. Коли обчислення необхідно повторити для набору вхідних значень, природно і вигідно представляти дані у вигляді масивів, а обчислення - з точки зору операцій з масивами. Обчислення, сформульовані таким чином, називаються *векторизованими*. Векторизоване обчислення усуває необхідність у багатьох явних циклах над елементами масиву шляхом застосування пакетних операцій до даних масиву. Результатом є стислий і більш підтримуваний код, який дозволяє делегувати реалізацію (наприклад, поелементних операцій з масивами) до більш ефективних бібліотек низького рівня.

Наприклад, порівняйте множення двох одновимірних списків з n чисел, a і b, в ядрі мови Python:

```
c = []
for i in range(n):
    c.append(a[i] * b[i])
```

та за допомогою масивів NumPy:

```
c = a * b
```

Тому векторизовані обчислення можуть бути значно швидшими, ніж послідовні обчислення по елементах. Це особливо важливо в інтерпретованій мові, такій як Python, де цикл по елементах тягне за собою значні накладні витрати на продуктивність [12].

У науковому обчислювальному середовищі Python ефективні структури даних для роботи з масивами надаються бібліотекою NumPy. Ядро NumPy реалізовано на C і забезпечує ефективні функції для маніпулювання та обробки масивів. На перший погляд, масиви NumPy мають деяку схожість зі структурою даних списку Python. Але важливою відмінністю є те, що хоча списки Python є загальними контейнерами об’єктів, масиви NumPy є однорідними та типізованими масивами фіксованого розміру. Однорідний означає, що всі елементи в масиві мають однаковий тип даних. Фіксований розмір означає, що розмір масиву неможливо змінити (без створення нового масиву). З цих та інших причин операції та функції, що діють на масивах NumPy, можуть бути набагато ефективнішими, ніж ті, що використовують списки Python.

NumPy - це основна бібліотека для наукових обчислень на Python. Вона надає високопродуктивний багатовимірний об’єкт масиву та інструменти для роботи з цими масивами. У Python ми маємо списки, які служать цілям масивів,

але вони повільно обробляються. NumPy має на меті надати об’єкт масиву, який до 50 разів швидше традиційних списків Python. Об'єкт масиву в NumPy називається ndarray, він надає багато допоміжних функцій, які роблять роботу з ndarray дуже легкою. Масиви дуже часто використовуються в науці про дані, де швидкість та ресурси дуже важливі. Більше інформації про NumPy можна знайти на веб-сайті www.numpy.org.

Ви можете встановити NumPy через командний рядок за допомогою такої команди:

```
pip install numpy
```

Імпорт модулів

Для того, щоб використовувати бібліотеку NumPy, нам потрібно імпортувати її до нашої програми. За домовленістю, модуль numpy імпортують під псевдонімом np:

```
import numpy as np
```

Зокрема, ця угода використовується в документації NumPy і в ширшій науковій екосистемі Python (SciPy, Pandas і так далі). Після цього ми можемо отримати доступ до функцій і класів у модулі numpy за допомогою простору імен np. У цьому розділі ми припускаємо, що модуль NumPy імпортується таким чином.

Об'єкт масиву NumPy

Ядром бібліотеки NumPy є структури даних для представлення багатовимірних масивів однорідних даних (з однаковим типом даних). Основною структурою даних для багатовимірних масивів у NumPy є клас ***ndarray***. Окрім даних, що зберігаються в масиві, ця структура даних також містить важливі метадані про масив, такі як його форма, розмір, тип даних та інші атрибути. Більш детальний опис цих атрибутів дивись у Таблиці 2-1. Повний список атрибутів з описами доступний у рядку docstring ndarray, доступ до якого можна отримати, викликавши довідку (np.ndarray) у інтерпретаторі Python.

Таблиця 2-1 Основні атрибути класу ndarray

Атрибут	Опис
shape	Кортеж, який містить кількість елементів (тобто довжину) для кожного виміру (осі) масиву.
size	Загальна кількість елементів у масиві.
ndim	Кількість розмірів (осей).
nbytes	Кількість байтів, використаних для зберігання даних.
data	"Буфер" в пам'яті, що містить фактичні елементи масиву
dtype	Тип даних елементів у масиві.

Створення масиву

Базовий метод створення масиву

Найпростіший спосіб створити невеликий масив NumPy - викликати функцію np.array зі списком або кортежем значень:

```
>>> array1d = np.array([1, 2, 3, 4])
>>> print(array1d)
```

```
[1 2 3 4]
>>> print(type(array1d))
<class 'numpy.ndarray'>
>>> print(array1d.shape)
(4,)
>>> b = np.array( [[1.,2.], [3.,4.]] )
>>> b
array([[1., 2.],
       [3., 4.]])
>>>
```

Індексування багатовимірного масиву NumPy дещо відрізняється від індексування звичайного списку списків Python: замість b [i][j] звертайтесь до індексу необхідного елемента як кортежа цілих чисел, b [i, j]:

```
>>> b[0,1]   # same as b[(0,1)]
2.0
>>> b[1,1] = 0.  # also for assignment
>>> b
array([[1., 2.],
       [3., 0.]])
>>>
```

Якщо ваш масив великий або ви не знаєте значення елементів під час створення, існує кілька способів оголосити масив певної форми, заповнений стандартними або довільними значеннями. Найпростіший і найшвидший, np.empty, приймає кортеж форми масиву і створює масив без ініціалізації його елементів: значення початкових елементів невизначені (як правило, випадкове сміття, взяте з будь-якого вмісту пам'яті, виділеного Python для масиву).

```
>>> np.empty((3,3))
array([[0.00e+000, 0.00e+000, 0.00e+000],
       [0.00e+000, 0.00e+000, 3.22e-321],
       [0.00e+000, 0.00e+000, 0.00e+000]])
>>>
```

Існують також допоміжні методи np.zeros та np.ones, які створюють масив зазначеної форми з елементами, попередньо заповненими 0 та 1 відповідно. np.empty, np.zeros і np.ones також беруть необов'язковий аргумент dtype.

```
>>> np.zeros((3,2))
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
>>> np.ones((3,3), dtype=int)
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]])
>>>
```

Ініціалізація масиву з послідовності

Для створення масиву, що містить послідовність чисел, є два методи: np.arange та np.linspace. np.arange є еквівалентом range NumPy, за винятком того, що він може генерувати послідовності з плаваючою точкою. Він також фактично виділяє пам'ять для елементів у ndarray замість повернення об'єкта, схожого на генератор.

```
>>> np.arange(7)
```

```
array([0, 1, 2, 3, 4, 5, 6])
>>> np.arange(1.5, 3., 0.5)
array([1.5, 2. , 2.5])
>>>
```

Як і range, масив, створений у цих прикладах, не включає останні елементи, 7 та 3. Однак у arange є проблема: через скінченну точність арифметики з плаваючою точкою не завжди можна дізнатися, скільки елементів буде створено. З цієї причини, а також тому, що часто потрібен останній елемент заданої послідовності, функція np.linspace може бути більш корисним способом створення послідовності. Наприклад, для створення рівномірно розподіленого масиву з п’яти чисел від 1 до 20 включно:

```
>>> np.linspace(1, 20, 5)
array([ 1., 5.75, 10.5, 15.25, 20.])
>>>
```

np.linspace містить пару необов'язкових boolean аргументів. По-перше, якщо для параметра *retstep* встановлено значення True, повертається інтервал між значеннями (розмір кроку):

```
>>> x, dx = np.linspace(0., 2*np.pi, 100, retstep=True)
>>> dx
0.06346651825433926
>>>
```

Це позбавить вас від окремого обчислення dx = (end-start)/(num-1); у цьому прикладі 100 точок між 0 і 2π включно рознесені на 2π /99 = 0.0634665… Нарешті, встановлення *endpoint* на False опускає кінцеву точку в послідовності, як для np.arange:

```
>>> x = np.linspace(0, 5, 5, endpoint=False)
>>> x
array([0., 1., 2., 3., 4.])
```

Зауважте, що масив, створений np.linspace, має dtype для чисел з плаваючою точкою, навіть якщо послідовність генерує цілі числа.

Ініціалізація масиву з функції

Для створення масиву, ініціалізованого значеннями, обчисленими за допомогою функції, використовуйте NumPy метод np.fromfunction, який бере в якості аргументів функцію та кортеж, що представляє форму потрібного масиву. Сама функція повинна приймати таку саму кількість аргументів, що і розміри в масиві: ці аргументи індексують кожен елемент, для якого функція повертає значення. Приклад зробить це більш зрозумілим:

```
>>> def f(i, j):
        return 2 * i * j

>>> np.fromfunction(f,(4,3))
array([[ 0.,  0.,  0.],
       [ 0.,  2.,  4.],
       [ 0.,  4.,  8.],
       [ 0.,  6., 12.]])
>>>
```

Функція f викликається для кожного індексу у зазначеній формі, і значення, які вона повертає, використовуються для ініціалізації відповідних

елементів. В попередньому прикладі при бажанні можна використати анонімну лямбда-функцію:

```
np.fromfunction(lambda i,j: 2*i*j, (4,3))
```

Універсальні функції (ufuncs)

На додаток до основних арифметичних операцій додавання, ділення тощо, NumPy надає багато знайомих математичних функцій, які виконує math модуль, реалізованих у вигляді так званих універсальних функцій, які діють на кожен елемент масиву, створюючи масив без необхідність явного циклу. Універсальні функції - це спосіб векторизації NumPy, що сприяє чистому, ефективному та простому у обслуговуванні коду. Наприклад,

```
>>> x = np.linspace(1, 5, 5)
>>> x**2
array([ 1.,  4.,  9., 16., 25.])
>>> x - 1
array([0., 1., 2., 3., 4.])
>>> np.sqrt(x - 1)
array([0. , 1. , 1.41421356, 1.73205081, 2. ])
>>> y = np.exp(-np.linspace(0., 2., 5))
>>> np.sin(x - y)
array([0., 0.98431873, 0.48771645, -0.59340065, -0.98842844])
>>>
```

Множення масивів відбувається поелементно (elementwise): множення матриць реалізується оператором @ або функцією dot NumPy:

```
>>> a = np.array( ((1, 2), (3, 4)) )
>>> b = a
>>> a * b  # elementwise multiplication
array([[ 1,  4],
       [ 9, 16]])
>>> a @ b  # matrix multiplication; also a.dot(b) or np.dot(a, b)
array([[ 7, 10],
       [15, 22]])
```

Оператори порівняння та логіки (~, & та | для not, and та or, відповідно) також векторизовані і призводять до масивів булевих значень:

```
>>> a = np.linspace(1, 6, 6)**3
>>> print(a)
[  1.   8.  27.  64. 125. 216.]
>>> print(a > 100)
[False False False False  True  True]
>>> print((a < 10) | (a > 100))
[ True  True False False  True  True]
```

Спеціальні значення NumPy, nan та inf

NumPy визначає два особливих значення для представлення результатів обчислень, які не визначені математично або не кінцеві. Значення np.nan ("Не число", NaN) представляє результат обчислення, яке не є чітко визначеною математичною операцією (наприклад, 0/0); np.inf представляє нескінченність. Наприклад,

```
>>> a = np.arange(4, dtype='f8')
>>> a /= 0
... RuntimeWarning: invalid value encountered in true_divide ...
... RuntimeWarning: divide by zero encountered in true_divide ...
```

```
>>> a
array([nan, inf, inf, inf])
```

Приклад. Магічний квадрат - це сітка чисел N × N, в якій сума чисел в кожному рядку, стовпці та основній діагоналі складають однакову величину (рівну N (N^2 + 1) /2).

Спосіб побудови магічного квадрата для непарного N виглядає наступним чином:

Крок 1. Почніть із середини верхнього ряду і нехай n = 1.

Крок 2. Вставте n у поточне положення сітки.

Крок 3. Якщо n = N^2 сітка завершена, зупиніться. В іншому випадку приріст n.

Крок 4. Переміщення по діагоналі вгору і вправо, перенесення до першого стовпця або останнього рядка, якщо переміщення веде поза сітки. Якщо ця клітинка вже заповнена, перемістіться вертикально вниз на один пробіл.

Крок 5. Поверніться до кроку 2.

Наступна програма створює та відображає магічний квадрат [13].

```
# Create an N x N magic square. N must be odd.
import numpy as np

N = 5
magic_square = np.zeros((N, N), dtype=int)
n = 1
i, j = 0, N//2
while n <= N**2:
    magic_square[i, j] = n
    n += 1
    newi , newj = (i - 1) % N, (j + 1)% N
    if magic_square[newi , newj]:
        i += 1
    else:
        i, j = newi , newj
print(magic_square)
```

На виході отримаємо магічний квадрат 5 × 5:

```
[[17 24  1  8 15]
 [23  5  7 14 16]
 [ 4  6 13 20 22]
 [10 12 19 21  3]
 [11 18 25  2  9]]
```

Читання та запис масиву у файл

Наукові дані часто зчитуються з текстового файлу, який може містити коментарі, відсутні значення та порожні рядки. Стовпці значень можуть бути вирівняні у форматі фіксованої ширини або розділені одним або кількома символами-розмежувачами (наприклад, пробілами, табуляціями чи комами). Крім того, у файлі може бути описовий заголовок і навіть виноски, що ускладнює аналіз безпосередньо за допомогою рядкових методів Python.

NumPy надає кілька функцій для зчитування даних з текстового файлу. Найпростіша, np.loadtxt, обробляє багато поширених випадків; більш складна np.genfromtxt дозволяє краще обробляти відсутні значення та колонтитули.

np.save та np.load

Існує незалежний від платформи двійковий формат для збереження масиву NumPy. Команда

```
np.save('my-array.npy', a)
```

збереже масив `a` у двійковому файлі my-array.npy (розширення .npy додається, якщо воно не надається). Потім масив можна завантажити за допомогою NumPy на будь-якій іншій операційній системі

```
a = np.load('my-array.npy')
```

np.loadtxt

Прототип методу np.loadtxt має вигляд:

```
np.loadtxt(fname , dtype=<class 'float'>, comments='#',
           delimiter=None , converters=None , skiprows=0,
           usecols=None , unpack=False , ndmin=0)
```

Аргументи такі:

• fname: Єдиний необхідний аргумент, який може бути іменем файлу, відкритим файлом або генератором, що повертає рядки даних для аналізу.

• dtype: Тип даних масиву (за замовчуванням плаваючий).

• comments: Коментарі у файлі зазвичай починаються з якогось символу, наприклад # (як у Python) або %. За замовчуванням він встановлений на #.

• delimiter: рядок, що використовується для розділення стовпців даних у файлі; за замовчуванням це значення None, тобто будь-яка кількість пробілів (пробілів, табуляцій) розмежовує дані. Щоб прочитати файл, розділений комами (csv), встановіть delimiter= ','.

• converters: необов'язковий словник, що відображає індекс стовпця у функцію, що перетворює значення рядків у цьому стовпці в дані (наприклад, з плаваючою точкою).

• skiprows: ціле число, що дає кількість рядків на початку файлу для пропуску перед читанням даних (наприклад, для проходження рядків заголовка). За замовчуванням 0 (без заголовка).

• usecols: послідовність індексів стовпців, що визначає, які стовпці файлу повертати як дані; за замовчуванням це значення None, тобто всі стовпці будуть проаналізовані та повернуті.

• unpack: за замовчуванням таблиця даних повертається в одному масиві рядків і стовпців, що відображає структуру прочитаного файлу. Установка unpack = True буде транспонувати цей масив так, що окремі стовпці можна вибирати та призначати різним змінним.

• ndmin: Мінімальна кількість розмірів, які має мати повернутий масив. За замовчуванням 0 (тому файл, що містить один номер, читається як скаляр); його також можна встановити на 1 або 2.

Приклад. Збережіть масив [[1.20, 2.20, 3.00], [4.14, 5.65, 6.42]] у файлі з назвою my_array.txt і прочитайте його назад до змінної my_arr.

```
import numpy as np
```

```
arr = np.array([[1.20, 2.20, 3.00], [4.14, 5.65, 6.42]])
np.savetxt("my_arr.txt", arr, fmt="%.2f",\
           header = "Col1 Col2 Col3")
my_arr = np.loadtxt("my_arr.txt")
print(my_arr)
```

Матриці

Масиви NumPy також служать матрицями, які є фундаментальними в математиці та обчислювальному програмуванні. Матриця - це просто двовимірний масив.

Одинична матриця (розміру n) — це матриця n × n, де (i, i)-й елемент дорівнює 1, а (i, j)-ий елемент дорівнює нулю для i ≠ j. Існує функція створення масиву, яка дає одиничну матрицю n × n для заданого значення n:

```
np.eye(3)
# array([[1., 0., 0.],
#        [0., 1., 0.],
#        [0., 0., 1.]])
```

Транспонування матриці

Масиви NumPy можна легко транспонувати, викликавши метод transpose для об’єкта масиву. Насправді, оскільки це така поширена операція, масиви мають зручну властивість T, яка повертає транспоновану матрицю. Транспонування змінює порядок форми матриці (масиву), так що рядки стають стовпцями, а стовпці — рядками.

```
mat = np.array([[1, 2], [3, 4]])
mat.transpose()
# array([[1, 3],
#        [2, 4]])
mat.T
# array([[1, 3],
#        [2, 4]])
```

Матричне множення

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[-1, 1], [0, 1]])
A @ B
# array([[-1, 3],
#        [-3, 7]])
A * B # different from A @ B
# array([[-1, 2],
#        [ 0, 4]])
```

Детермінант і обернена матриця

Підпрограма NumPy для обчислення визначника матриці міститься в окремому модулі, який називається linalg. Цей модуль містить багато загальних процедур для лінійної алгебри, яка є розділом математики, що охоплює векторну та матричну алгебру. Підпрограмою для обчислення визначника квадратної матриці є програма det:

```
from numpy import linalg
linalg.det(A) # -2.0000000000000004
```

Оберненою до n × n матриці A є (обов'язково єдина) n × n матриця B, така що AB = BA = I, де I позначає одиничну n × n матрицю, а множення, що виконується тут, є множенням матриць.

Підпрограма inv з модуля linalg обчислює обернену матрицю, якщо вона існує:

```
linalg.inv(A)
# array([[-2. , 1. ],
#        [ 1.5, -0.5]])
```

Ми можемо перевірити чи справді матриця, обчислена підпрограмою inv, є матрицею, оберненою до A, помноживши матрицю A (з будь-якого боку) на обернену і перевіривши, що ми отримуємо ідентичну матрицю 2 × 2:

```
Ainv = linalg.inv(A)
Ainv @ A
# Approximately
# array([[1., 0.],
#        [0., 1.]])
A @ Ainv
# Approximately
# array([[1., 0.],
#        [0., 1.]])
```

Пакет linalg також містить ряд інших методів, таких як norm, який обчислює різні норми матриці. Він також містить функції для розкладання матриць різними способами та розв'язування систем рівнянь.

Системи рівнянь

Для ілюстрації методики розв'яжемо наприклад таку систему рівнянь:

$$\begin{aligned} 3x_1 - 2x_2 + x_3 &= 7 \\ x_1 + x_2 - 2x_3 &= -4 \\ -3x_1 - 2x_2 + x_3 &= 1 \end{aligned}$$

Ця система рівнянь має три невідомі значення: x_1, x_2 і x_3. Спочатку створюємо матрицю коефіцієнтів A і вектор b. Оскільки ми використовуємо NumPy як наш засіб роботи з матрицями та векторами, ми створюємо двовимірний масив NumPy для матриці A та одновимірний масив для b:

```
import numpy as np
from numpy import linalg
A = np.array([[3, -2, 1], [1, 1, -2], [-3, -2, 1]])
b = np.array([7, -4, 1])
```

Тепер рішення системи рівнянь можна знайти за допомогою процедури solve:

```
linalg.solve(A, b) # array([ 1., -1., 2.])
```

Чисельне розв'язування простих диференціальних рівнянь

Диференціальні рівняння виникають у ситуаціях, коли величина розвивається, як правило, з часом відповідно до заданого співвідношення. Вони надзвичайно поширені в інженерії та фізиці і з'являються цілком природно. Одним із класичних прикладів (дуже простого) диференціального рівняння є закон охолодження, розроблений Ньютоном. Температура тіла охолоджується зі швидкістю, пропорційною поточній температурі. Математично це означає, що

ми можемо записати похідну температури T тіла в момент t > 0 за допомогою диференціального рівняння

$$\frac{dT}{dt} = -kT$$

де k – додатна константа, що визначає швидкість охолодження. Це диференціальне рівняння можна розв'язати аналітично, спочатку «розділивши змінні», а потім інтегрувавши та перегрупувавши. Після виконання цієї процедури отримуємо загальне рішення

$$T(t) = T_0 e^{-kt}$$

де T_0 - початкова температура.

У цьому підрозділі ми розв'яжемо просте звичайне диференціальне рівняння чисельно, використовуючи підпрограму solve_ivp від SciPy.

Ми продемонструємо техніку чисельного розв'язування диференціального рівняння в Python за допомогою рівняння охолодження, описаного раніше, оскільки в цьому випадку ми можемо обчислити істинне рішення. Приймемо початкову температуру $T_0 = 50$ і $k = 0{,}2$. Давайте також знайдемо рішення для значень t від 0 до 5.

Загальне (першого порядку) диференціальне рівняння має вигляд

$$\frac{dy}{dt} = f(t, y)$$

де f деяка функція від t (незалежна змінна) і y (залежна змінна). У цій формулі T є залежною змінною, а f(t, T) = -kt. Підпрограми для розв'язування диференціальних рівнянь у пакеті SciPy вимагають функції f і початкового значення y_0 і діапазону значень t, де нам потрібно обчислити рішення. Для початку нам потрібно визначити нашу функцію f у Python і створити діапазон змінних y_0 і t, готовий для надання в підпрограму SciPy:

```
def f(t, y):
    return -0.2*y
t_range = (0, 5)
```

Далі нам потрібно визначити початкову умову, з якої слід знайти рішення. З технічних причин початкові значення y повинні бути вказані як одновимірний масив NumPy:

```
T0 = np.array([50.])
```

Оскільки в цьому випадку ми вже знаємо точне рішення, ми також можемо визначити його в Python, готове до порівняння з чисельним рішенням, яке ми будемо обчислювати:

```
def true_solution(t):
    return 50.*np.exp(-0.2*t)
```

Ми використовуємо процедуру solve_ivp з модуля integrate в SciPy для чисельного розв'язання диференціального рівняння. Ми додаємо параметр для максимального розміру кроку зі значенням 0,1, тому розв'язок обчислюється з розумною кількістю точок.

Далі ми витягуємо значення рішення з об'єкта sol, поверненого методом solve_ivp.

Оскільки ми також збираємося побудувати помилку апроксимації на тому самому малюнку, ми створюємо два підграфіки, використовуючи процедуру subplots.

В результаті отримаємо таку програму:

```
import numpy as np
from scipy import integrate
import matplotlib.pyplot as plt

def f(t, y):
    return -0.2*y

t_range = (0, 5)

T0 = np.array([50.])

def true_solution(t):
    return 50.*np.exp(-0.2*t)

sol = integrate.solve_ivp(f, t_range, T0, max_step=0.1)

t_vals = sol.t
T_vals = sol.y[0, :]

fig, (ax1, ax2) = plt.subplots(1, 2, tight_layout=True)

ax1.plot(t_vals, T_vals)
ax1.set_xlabel("$t$")
ax1.set_ylabel("$T$")
ax1.set_title("Solution of the cooling equation")

err = np.abs(T_vals - true_solution(t_vals))
ax2.semilogy(t_vals, err)
ax2.set_xlabel("$t$")
ax2.set_ylabel("Error")
ax2.set_title("Error in approximation")

plt.show()
```

Виконавши програму, отримаємо наступні графіки:

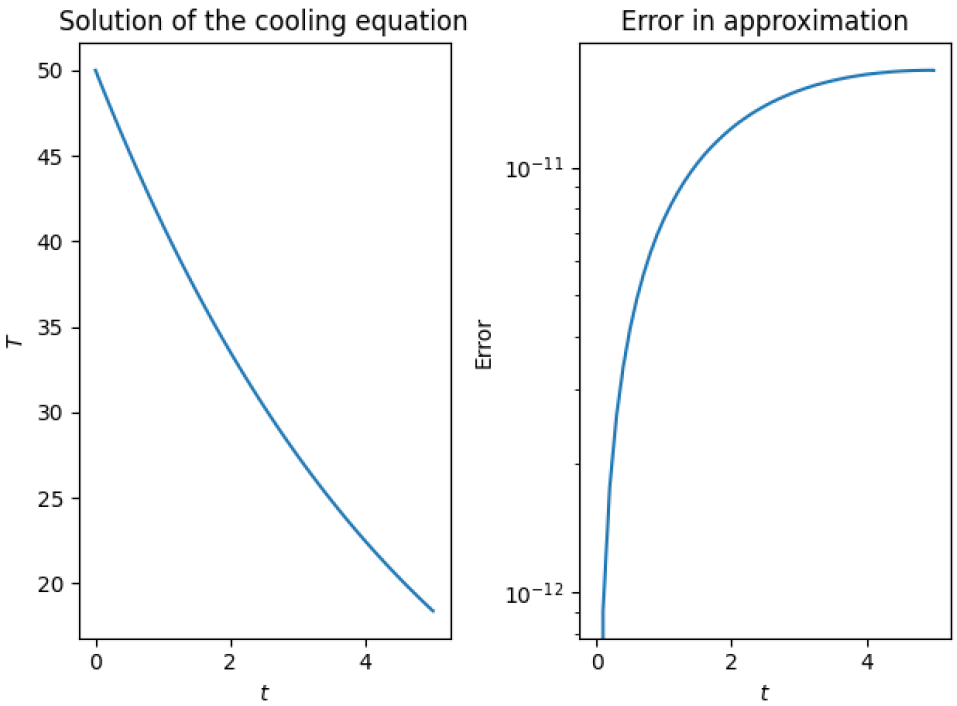

Рис. 21 Графік чисельного рішення

Лівосторонній графік на Рис. 21 показує зниження температури з часом, тоді як правий графік показує, що помилка збільшується, коли ми віддаляємося від відомого значення, заданого початковою умовою.

Створення ігр за допомогою Pygame

Pygame — це кросплатформна безкоштовна бібліотека Python з відкритим вихідним кодом, призначена для полегшення створення мультимедійних програм, таких як ігри. Розробка Pygame почалася ще в жовтні 2000 року, а версія Pygame 1.0 була випущена через півроку [16].

Pygame побудовано на основі бібліотеки SDL. SDL (або Simple Directmedia Layer) — це міжплатформна бібліотека розробки, призначена для забезпечення доступу до аудіо, клавіатури, миші, джойстика та графічного обладнання через OpenGL і Direct3D.

SDL офіційно підтримує Windows, Mac OS X, Linux, iOS і Android. Сама SDL написана на C, а Pygame надає обгортку навколо SDL. Однак Pygame додає функціональні можливості, яких немає в SDL, щоб полегшити створення графічних або відеоігор. Ці функції включають векторну математику, виявлення зіткнень, керування графіком 2D-спрайтів, підтримку MIDI, камеру, маніпуляції з масивом пікселів, трансформації, фільтрацію, розширену підтримку шрифтів та малювання.

Інсталяція Pygame

Щоб інсталювати Pygame, введіть таку команду в командному рядку термінала:

```
pip install pygame
```

Ця команда запускає менеджер пакетів pip і той встановлює пакет pygame до інсталяції Python.

Ключові поняття Pygame

Далі в цьому розділі ми ознайомимось з ключовими поняттями, модулі, класами та функціями Pygame та створимо дуже простий перший додаток Pygame. У наступному розділі відбудеться розробка простої відеоігри в аркадному стилі, яка ілюструє, як можна створити гру за допомогою Pygame.

Поверхня відображення

Дисплейна поверхня (Display Surface) є найважливішою частиною гри Pygame. Це головне вікно вашої гри, яке може бути будь-якого розміру, однак ви можете мати лише одну дисплейну поверхню.

Багато в чому дисплейна поверхня схожа на чистий аркуш паперу, на якому можна малювати. Сама поверхня складається з пікселів, які пронумеровані від 0,0 у верхньому лівому куті, а розташування пікселів проіндексовано по осі x та осі y. Це показано нижче:

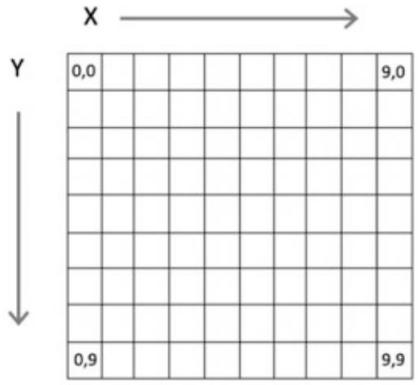

Рис. 22 Display Surface

Дисплейна поверхня створюється за допомогою функції pygame.display.set_mode(). Ця функція приймає кортеж, який визначає розміри поверхні. Наприклад:

```
display_surface = pygame.display.set_mode((400, 300))
```

Це створить дисплейну поверхню (вікно) розміром 400 на 300 пікселів.

Після того, як у вас є дисплейна поверхня, ви можете заповнити її відповідним кольором фону (за замовчуванням чорний), однак, якщо ви хочете інший колір фону або хочете очистити все, що раніше було намальовано на поверхні, ви можете використовувати поверхні метод fill():

```
WHITE = (255, 255, 255)
display_surface.fill(WHITE)
```

Щоб підвищити продуктивність, будь-які зміни, які ви вносите до дисплейної поверхні, насправді відбуваються у фоновому режимі і не відображатимуться на фактичному дисплеї, який бачить користувач, доки ви не викличете методи update() або flip() на поверхні. Наприклад:

- `pygame.display.update()`
- `pygame.display.flip()`

Метод update() перемалює дисплей з усіма змінами, внесеними в дисплей у фоновому режимі. Він має додатковий параметр, який дозволяє вказати лише область дисплея для оновлення (це визначається за допомогою Rect, який представляє прямокутну область на екрані). Метод flip() завжди оновлює весь дисплей (і таким чином робить те саме, що й метод update() без параметрів).

Інший метод, який не є спеціально методом поверхні відображення, але який часто використовується під час створення поверхні відображення, забезпечує заголовок для вікна верхнього рівня. Це функція pygame.display.set_caption(). Наприклад:

```
pygame.display.set_caption('Hello World')
```

Події

Подібно до того, як системи графічного інтерфейсу користувача, описані в попередніх розділах, мають цикл подій, який дозволяє програмісту визначити, що робить користувач (у цих випадках це, як правило, вибір пункту меню, натискання кнопки або введення даних, тощо); Pygame має цикл подій, який дозволяє грі визначити, що робить гравець. Наприклад, користувач може натиснути клавішу зі стрілкою вліво або вправо. Це представлено подією.

Типи подій

Кожна подія, що відбувається, має пов'язану інформацію, наприклад тип цієї події. Наприклад:

• Натискання клавіші призведе до події типу KEYDOWN, а відпускання клавіші призведе до типу події KEYUP.

• Вибір кнопки закриття вікна призведе до створення типу події QUIT, тощо.

• Використання миші може генерувати події MOUSEMOTION, а також типи подій MOUSEBUTTONDOWN і MOUSEBUTTONUP.

• За допомогою джойстика можна генерувати кілька різних типів подій, включаючи JOYAXISMOTION, JOYBALLMOTION, JOYBUTTONDOWN і JOYBU TTONUP.

Ці типи подій повідомляють вам, що сталося. Це означає, що ви можете вибрати типи подій, з якими ви хочете мати справу, і ігнорувати інші події.

Інформація про подію

Кожен тип об'єкта події надає інформацію, пов'язану з цією подією. Наприклад, об'єкт події, орієнтований на клавішу, надасть фактично натиснуту клавішу, тоді як об'єкт події, орієнтований на мишу, надасть інформацію про положення миші, яку кнопку було натиснуто тощо. Якщо ви спробуєте отримати доступ до атрибута для події, яка цей атрибут не підтримує, то буде згенеровано помилку.

Нижче наведено деякі атрибути, доступні для різних типів подій:

• KEYDOWN та KEYUP події мають атрибут key та атрибут mod (що вказує, чи натискаються інші клавіші модифікації, наприклад Shift).

• MOUSEBUTTONUP і MOUSEBUTTONDOWN мають атрибут pos, який містить кортеж, що вказує розташування миші за координатами x і y на нижній поверхні. Вони також мають атрибут button, що вказує, яка кнопка миші була натиснута.

• MOUSEMOTION має атрибути pos, rel і buttons. Pos — це кортеж, що вказує розташування курсору миші по x і y. Атрибут rel вказує на кількість переміщень миші, а buttons вказують на стан кнопок миші.

Нариклад, якщо ми хочемо перевірити тип події клавіатури, а потім перевірити, чи натиснута клавіша була пробілом, ми можемо написати:

```
if event.type == pygame.KEYDOWN:
    # Check to see which key is pressed
    if event.key == pygame.K_SPACE:
        print('space')
```

Існує багато констант клавіатури, які використовуються для представлення клавіш на клавіатурі в Pygame. Константа K_SPACE, використана вище, є лише однією з них.

Усі константи клавіатури мають префікс «K_», за яким слідує клавіша або назва клавіші, наприклад:

• K_TAB, K_SPACE, K_PLUS, K_0, K_1, K_AT, K_a, K_b, K_z, K_DELTE, K_DOWN, K_LEFT, K_RIGHT, K_LEFT тощо.

Додаткові константи клавіатури надаються для станів модифікаторів, які можна об'єднати з наведеними вище, наприклад KMOD_SHIFT, KMOD_CAPS, KMOD_CTRL та KMOD_ALT.

Черга подій

Події надходять до програми Pygame через чергу подій.

Черга подій використовується для збору подій, коли вони відбуваються. Наприклад, припустимо, що користувач двічі клацає мишею та двічі клавішею, перш ніж програма зможе їх обробити; тоді в черзі подій буде чотири події, як показано нижче:

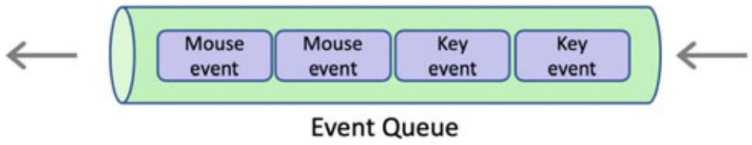

Рис. 22 Черга подій

Потім програма може отримати ітератор із черги подій і обробляти події по черзі. Поки програма обробляє ці події, можуть відбуватися інші події, які будуть додані до черги подій. Коли програма завершила обробку початкової колекції подій, вона може отримати наступний набір подій для обробки.

Однією з істотних переваг цього підходу є те, що жодна подія ніколи не втрачається; тобто якщо користувач двічі клацне мишею, поки програма обробляє попередній набір подій; вони будуть записані та додані до черги подій. Ще одна перевага полягає в тому, що події будуть представлені програмі в тому порядку, в якому вони відбувалися.

Функція pygame.event.get() прочитає всі події, які наразі знаходяться в черзі подій (видаляючи їх із черги подій). Метод повертає EventList, який є ітераційним списком прочитаних подій. Потім кожну подію можна обробляти по черзі. Наприклад:

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        print('Received Quit Event:')
    elif event.type == pygame.MOUSEBUTTONDOWN:
        print('Received Mouse Event')
    elif event.type == pygame.KEYDOWN:
        print('Received KeyDown Event')
```

Перша програма Pygame

Програма, яку ми створимо, відобразить вікно Pygame з назвою «Hello World». Тоді ми зможемо вийти з гри. Хоча технічно це не гра, але вона має базову архітектуру програми Pygame.

```
import pygame
def main():
    print('Starting Game')
    print('Initialising Pygame')
    pygame.init() # Required by every Pygame application
    print('Initialising HelloWorldGame')
    pygame.display.set_mode((200, 100))
    pygame.display.set_caption('Hello World')
    print('Update display')
    pygame.display.update()
    print('Starting main Game Playing Loop')
    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                print('Received Quit Event:', event)
                running = False
    print('Game Over')
    pygame.quit()
if __name__ == '__main__':
    main()
```

Більш інтерактивна програма Pygame

Перша програма Pygame, яку ми розглянули раніше, просто показувала вікно з написом «Hello World». Тепер ми можемо трохи розширити її, погравши з деякими функціями, які ми розглянули вище.

Нова програма додасть деяку обробку подій миші. Це дозволить нам визначити розташування миші, коли користувач клацнув на вікні, і намалювати в цій точці невелике синє поле.

Якщо користувач клацне мишею кілька разів, ми отримаємо кілька синіх прямокутників. Це показано нижче.

Рис. 23 Вікно другої програми

Це все ще не справжня гра, але цей застосунок Pygame є більш інтерактивним.

Вихідний код програми представлений нижче:

```
import pygame
FRAME_REFRESH_RATE = 30
```

```
BLUE = (0, 0, 255)
BACKGROUND = (255, 255, 255) # White
WIDTH = 10
HEIGHT = 10
def main():
    print('Initialising PyGame')
    pygame.init() # Required by every PyGame application
    print('Initialising Box Game')
    display_surface = pygame.display.set_mode((400, 300))
    pygame.display.set_caption('Box Game')
    print('Update display')
    pygame.display.update()
    print('Setup the Clock')
    clock = pygame.time.Clock()
    # Clear the screen of current contents
    display_surface.fill(BACKGROUND)
    print('Starting main Game Playing Loop')
    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                print('Received Quit Event:', event)
                running = False
            elif event.type == pygame.MOUSEBUTTONDOWN:
                print('Received Mouse Event', event)
                x, y = event.pos
                pygame.draw.rect(display_surface, BLUE, [x, y,
WIDTH, HEIGHT])
        # Update the display
        pygame.display.update()
        # Defines the frame rate - the number of frames per
second
        # Should be called once per frame (but only once)
        clock.tick(FRAME_REFRESH_RATE)
    print('Game Over')
    # Now tidy up and quit Python
    pygame.quit()
if __name__ == '__main__':
    main()
```

Гра StarshipMeteors

У цьому підрозділі ми створимо гру, в якій ви керуєте зоряним кораблем через поле метеорів [16]. Чим довше ви граєте в гру, тим більшу кількість метеорів ви зустрінете. Типовий екран гри показаний нижче:

Рис. 24 Типовий екран гри

Ми реалізуємо кілька класів для представлення сутностей у грі. Використання класів не є обов’язковим способом реалізації гри, і слід зазначити, що багато розробників уникають використання класів. Однак використання класу дозволяє зберігати дані, пов’язані з об’єктом у грі, в одному місці; це також спрощує створення кількох екземплярів одного і того ж об'єкта (наприклад, метеорів) у грі.

Нижче показано класи та їх взаємозв’язки:

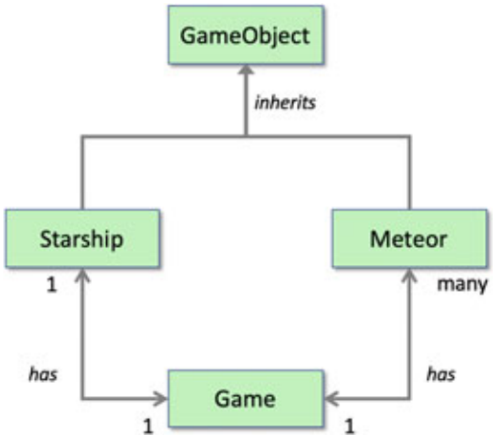

Рис. 25 Діаграма класів

Ця діаграма показує, що класи Starship і Meteor розширюють клас під назвою GameObject.

У свою чергу, це також показує, що Game має співвідношення 1:1 з класом Starship. Тобто Game містить посилання на один Starship, а Starship, у свою чергу, містить одне посилання на Game.

На противагу цьому Game має відношення 1 до багатьох із класом Meteor. Тобто об’єкт Game містить посилання на багато Meteors, і кожен Meteor містить посилання на один об’єкт Game.

Клас GameObject

Клас GameObject визначає три методи:

Метод load_image() можна використовувати для завантаження зображення, яке буде використано для візуального представлення конкретного типу ігрового об’єкта. Потім метод використовує ширину та висоту зображення для визначення ширини та висоти ігрового об’єкта.

Метод rect() повертає прямокутник, що представляє поточну область, яку використовує ігровий об'єкт на базовій поверхні малювання. Прямокутники дуже корисні для порівняння розташування одного об'єкта з іншим (наприклад, коли визначають, чи відбулося зіткнення).

Метод draw() малює зображення GameObjects на display_surface, використовуючи поточні координати x і y. Метод можна замінити в підкласах, якщо їх потрібно намалювати іншим способом.

Код для класу GameObject представлено нижче:

```
class GameObject:
    def load_image(self, filename):
        self.image = pygame.image.load(filename).convert()
        self.width = self.image.get_width()
        self.height = self.image.get_height()
    def rect(self):
        """ Generates a rectangle representing the objects
            location and dimensions """
        return   pygame.Rect(self.x,   self.y,   self.width,
self.height)
    def draw(self):
        """ draw the game object at the
            current x, y coordinates """
        self.game.display_surface.blit(self.image,    (self.x,
self.y))
```

Клас GameObject безпосередньо розширюється класами Starship і Meteor.

На даний момент існує лише два типи ігрових елементів: зоряний корабель і метеори; але в майбутньому це може бути поширено на планети, комети, падаючі зірки тощо.

Відображення Starship

Гравець-людина в цій грі керуватиме зоряним кораблем, який можна переміщати по дисплею.

Зоряний корабель буде представлений екземпляром класу Starship. Цей клас розширить клас GameObject, який містить загальну поведінку для будь-якого типу елементів, представлених у грі.

Клас Starship визначає власний метод __init__(), який бере посилання на game, частиною якої є зореліт. Цей метод ініціалізації встановлює початкове розташування Starship як половину ширини дисплея для координати x і висоти відображення мінус 40 для координати y (це дає трохи буфера перед кінцем екрана). Потім він використовує метод load_image() з батьківського класу GameObject, щоб завантажити зображення, яке буде використовуватися для представлення Starship. Це зображення зберігається у файлі під назвою starship.png. На даний момент ми залишимо клас Starship як він є (однак ми повернемося до цього класу, щоб перетворити його на рухомий об'єкт у наступному розділі).

Нижче наведена поточна версія класу Starship:

```
class Starship(GameObject):
    """ Represents a starship"""
    def __init__(self, game):
        self.game = game
```

```
        self.x = DISPLAY_WIDTH / 2
        self.y = DISPLAY_HEIGHT - 40
        self.load_image('starship.png')
```

У класі Game ми тепер додамо рядок до методу __init__() для ініціалізації об'єкта Starship. Ось цей рядок:

```
# Set up the starship
self.starship = Starship(self)
```

Ми також додамо рядок до основного циклу while у методі play() безпосередньо перед тим, як оновити дисплей. Цей рядок викличе метод draw() для об'єкта зоряного корабля:

```
# Draw the starship
self.starship.draw()
```

Це матиме ефект малювання зоряного корабля на поверхні малювання вікна у фоновому режимі до того, як дисплей буде оновлено.

Коли ми зараз запустимо цю версію гри StarshipMeteor, ми бачимо Starship на дисплеї:

Рис. 26 Відображення зоряного корраблля

Звичайно, зараз зоряний корабель не рухається; але ми розглянемо це в наступному розділі.

Переміщення космічного корабля

Ми хочемо мати можливість переміщати Starship в межах екрана дисплея. Для цього нам потрібно змінити його координати x і y у відповідь на натискання користувачами різних клавіш.

Ми будемо використовувати клавіші зі стрілками для переміщення вгору і вниз по екрану або ліворуч або праворуч від екрана. Для цього ми визначимо чотири методи в класі Starship; ці методи будуть рухати зореліт вгору, вниз, вліво і вправо тощо.

Оновлений клас Starship показано нижче:

```
class Starship(GameObject):
    """ Represents a starship"""
    def __init__(self, game):
        super().__init__()
        self.game = game
        self.x = DISPLAY_WIDTH / 2
        self.y = DISPLAY_HEIGHT - 40
        self.load_image('starship.png')
    def move_right(self):
```

```
            """ moves the starship right across the screen """
            self.x = self.x + STARSHIP_SPEED
            if self.x + self.width > DISPLAY_WIDTH:
                self.x = DISPLAY_WIDTH - self.width
        def move_left(self):
            """ Move the starship left across the screen """
            self.x = self.x - STARSHIP_SPEED
            if self.x < 0:
                self.x = 0
        def move_up(self):
            """ Move the starship up the screen """
            self.y = self.y - STARSHIP_SPEED
            if self.y < 0:
                self.y = 0
        def move_down(self):
            """ Move the starship down the screen """
            self.y = self.y + STARSHIP_SPEED
            if self.y + self.height > DISPLAY_HEIGHT:
                self.y = DISPLAY_HEIGHT - self.height
        def __str__(self):
            return 'Starship(' + str(self.x) + ', ' + str(self.y)
+ ')'
```

Ця версія класу Starship визначає різні методи переміщення. Ці методи використовують нове глобальне значення STARSHIP_SPEED, щоб визначити, як далеко і як швидко рухається Starship. Якщо ви хочете змінити швидкість, з якою рухається Starship, ви можете змінити це глобальне значення.

Залежно від наміченого напрямку нам потрібно буде змінити координати x або y зоряного корабля.

• Якщо зореліт рухається ліворуч, то координата x зменшується на STARSHIP_SPEED,

• якщо він рухається вправо, то координата x збільшується на STARSHIP_SPEED,

• у свою чергу, якщо зоряний корабель рухається вгору по екрану, то координата y зменшується на STARSHIP_SPEED,

• але якщо він рухається вниз по екрану, то координата y збільшується на STARSHIP_SPEED.

Звичайно, ми не хочемо, щоб наш Зоряний корабель відлетів від краю екрана, тому потрібно перевірити, чи досяг він меж екрана. Таким чином проводяться тести, щоб побачити, чи опустилися значення x або y нижче нуля чи вище значень DISPLAY_WIDTH або DISPLAY_HEIGHT. Якщо будь-яка з цих умов виконується, значення x або y скидаються до відповідного за замовчуванням.

Тепер ми можемо пов’язати ці методи з введенням гравця. Оскільки для цього ми використовуємо клавіші зі стрілками вліво, вправо, вгору та вниз (K_LEFT, K_RIGHT, K_UP і K_DOWN), ми можемо розширити цикл обробки подій, який ми вже визначили для основного циклу гри. Коли натиснуто одну з цих клавіш, ми викличемо відповідний метод переміщення на об’єкті зоряного корабля, що вже утримується об’єктом Game.

Основна обробка подій для циклу тепер наступна:

```
# Work out what the user wants to do
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        is_running = False
    elif event.type == pygame.KEYDOWN:
        # Check to see which key is pressed
        if event.key == pygame.K_RIGHT:
            # Right arrow key has been pressed
            # move the player right
            self.starship.move_right()
        elif event.key == pygame.K_LEFT:
            # Left arrow has been pressed
            # move the player left
            self.starship.move_left()
        elif event.key == pygame.K_UP:
            self.starship.move_up()
        elif event.key == pygame.K_DOWN:
            self.starship.move_down()
        elif event.key == pygame.K_p:
            self._pause()
        elif event.key == pygame.K_q:
            is_running = False
```

Проте ми не зовсім закінчили. Якщо ми спробуємо запустити цю версію програми, ми отримаємо слід зоряних кораблів, намальований на екрані; наприклад:

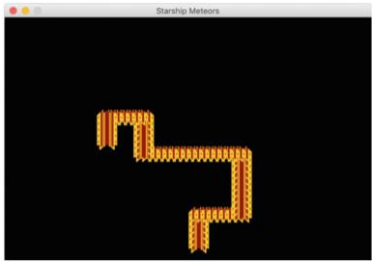

Рис. 27 Слід від зоряних кораблів

Проблема в тому, що ми перемальовуємо зоряний корабель в іншому положеннів той час як попереднє зображення все ще присутнє.

Тепер у нас є два варіанти, один – просто заповнити весь екран чорним; ефективно приховуючи все, що було намальовано до цього часу; або, як альтернативу, ми можемо просто перемалювати область, яка використовувалась для попередньої позиції зображення. Який підхід буде прийнятий, залежить від конкретного сценарію, представленого вашою грою. Оскільки після додавання на екрані буде багато метеорів; найпростіший варіант — перезаписати все на екрані перед тим, як перемалювати зореліт. Тому ми додамо наступний рядок:

```
# Clear the screen of current contents
self.display_surface.fill(BACKGROUND)
```

Цей рядок додається безпосередньо перед тим, як ми намалюємо Starship в головній грі в циклі while.

Тепер, коли ми переміщаємо Starship, старе зображення видаляється, перш ніж ми намалюємо нове зображення:

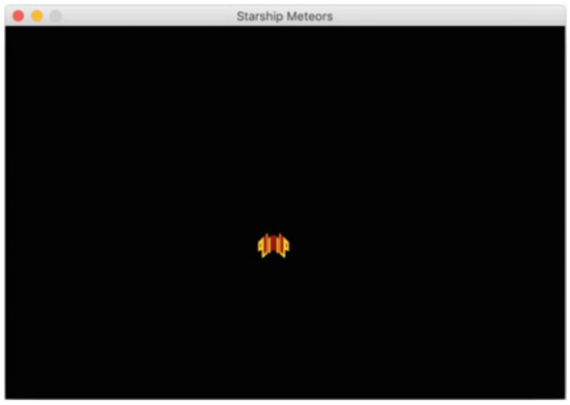

Рис. 28 Вікно гри після видалення старого зображення

Слід зазначити, що ми також визначили інше глобальне значення - BACKGROUND, яке використовується для збереження кольору фону ігрової поверхні. Воно встановлено в чорний колір, як показано нижче:

```
# Define default RGB colours
BACKGROUND = (0, 0, 0)
```

Якщо ви хочете використовувати інший колір фону, змініть це глобальне значення.

Додавання класу Meteor

Клас Meteor також буде підкласом класу GameObject. Однак він надасть лише метод move_down(), а не різноманітні методи переміщення Starship.

Він також повинен мати випадкову початкову координату x, щоб, коли до гри додається метеор, його початкова позиція змінювалася. Цю випадкову позицію можна створити за допомогою функції random.randint(), використовуючи значення від 0 до ширини поверхні малювання. Метеор також почнеться у верхній частині екрана, тому матиме іншу початкову координату у від зоряного корабля. Нарешті, ми також хочемо, щоб наші метеори мали різну швидкість; це може бути інше випадкове число від 1 до певної заданої максимальної швидкості метеора.

Щоб підтримати все це, нам потрібно додати до модулів, що імпортуються, random та визначити кілька нових глобальних значень, наприклад:

```
import pygame, random

INITIAL_METEOR_Y_LOCATION = 10
MAX_METEOR_SPEED = 5
```

Тепер ми можемо визначити клас Meteor:

```
class Meteor(GameObject):
    """ represents a meteor in the game """
    def __init__(self, game):
        super().__init__()
        self.game = game
        self.x = random.randint(0, DISPLAY_WIDTH)
        self.y = INITIAL_METEOR_Y_LOCATION
        self.speed = random.randint(1, MAX_METEOR_SPEED)
        self.load_image('meteor.png')
    def move_down(self):
        """ Move the meteor down the screen """
        self.y = self.y + self.speed
```

```
            if self.y > DISPLAY_HEIGHT:
                self.y = 5
        def __str__(self):
            return 'Meteor(' + str(self.x) + ', ' + str(self.y) +
')'
```

Метод __init__() для класу Meteor має ті самі кроки, що і в Starship; різниця в тому, що координата x і швидкість генеруються випадковим чином. Зображення, яке використовується для Meteor, також відрізняється, оскільки це «meteor.png».

Ми також реалізували метод move_down(). Це по суті те саме, що і в Starships move_down().

Зауважте, що на цьому етапі ми можемо створити підклас GameObject під назвою MoveableGameObject (який розширює GameObject) і перенести операції переміщення вгору до цього класу, а класи Meteor і Starship розширять цей клас. Однак ми насправді не хочемо дозволяти метеорам рухатися в будь-якому місці на екрані.

Тепер ми можемо додати метеори до класу Game. Ми додамо нове глобальне значення, щоб вказати кількість початкових метеорів у грі:

```
INITIAL_NUMBER_OF_METEORS = 8
```

Далі ми ініціалізуємо новий атрибут для класу Game, який міститиме список Meteors. Ми будемо використовувати тут список, оскільки хочемо збільшити кількість метеорів у міру просування гри.

Щоб полегшити цей процес, ми будемо використовувати list comprehension:

```
# Set up meteors
self.meteors = [Meteor(self) for _ in range(0,
INITIAL_NUMBER_OF_METEORS)]
```

Тепер у нас є список метеорів, які потрібно відобразити. Таким чином, нам потрібно оновити цикл while методу play(), щоб намалювати не тільки зореліт, але й усі метеори:

```
# Draw the meteors and the starship
self.starship.draw()
for meteor in self.meteors:
    meteor.draw()
```

Кінцевим результатом є те, що набір метеорних об’єктів створюється у випадкових початкових місцях у верхній частині екрана:

Рис. 29 Поява метеорів

Переміщення метеорів

Тепер ми хочемо мати можливість переміщати метеори вниз по екрану, щоб у Starship було кілька об'єктів, яких слід уникати.

Ми можемо зробити це дуже легко, оскільки ми вже реалізували метод move_down() у класі Meteor. Тому нам потрібно лише додати цикл for до основної гри в цикл while, який переміщуватиме всі метеори. Наприклад:

```
# Move the Meteors
for meteor in self.meteors:
    meteor.move_down()
```

Це можна додати після обробки події для циклу і до оновлення/перемальовування або оновлення екрана.

Тепер, коли ми запускаємо гру, метеори рухаються, і гравець може переміщатися зоряним кораблем між падаючими метеорами.

Рис. 30. Рух метеорів

Виявлення зіткнення

На даний момент гра буде грати вічно, оскільки немає кінцевого стану і немає спроб визначити, чи зіткнувся зоряний корабель з метеором.

Ми можемо додати виявлення зіткнення Meteor/Starship за допомогою PyGame Rects. Rect — це клас PyGame, який використовується для представлення прямокутних координат. Це особливо корисно, оскільки клас pygame.Rect надає кілька методів виявлення зіткнень, які можна використовувати для перевірки, чи знаходиться один прямокутник (або точка) всередині іншого прямокутника. Тому ми можемо використовувати один із методів, щоб перевірити, чи перетинається прямокутник навколо Starship з будь-яким із прямокутників навколо Метеорів.

Клас GameObject вже надає метод rect(), який повертає об'єкт Rect, що представляє поточний прямокутник об'єктів відносно поверхні малювання (по суті, поле навколо об'єкта, що відображає його розташування на екрані).

Таким чином, ми можемо написати метод виявлення зіткнень для класу Game, використовуючи rects, згенерований GameObject, і метод colliderect() класу Rect:

```
def _check_for_collision(self):
    """ Checks to see if any of the meteors have collided
        with the starship """
    result = False
```

```
        for meteor in self.meteors:
            if self.starship.rect().colliderect(meteor.rect()):
                result = True
                break
        return result
```

Зауважте, що тут ми дотримувались угоди щодо передування імені метода нижнього підкреслення, яка вказує, що цей метод слід вважати приватним для класу. Тому він ніколи не повинен викликатися нічим за межами класу Game. Ця угода визначена в PEP 8.

Тепер ми можемо використовувати цей метод в основному циклі while гри, щоб перевірити наявність зіткнення:

```
# Check to see if a meteor has hit the ship
if self._check_for_collision():
    starship_collided = True
```

Цей фрагмент коду також представляє нову локальну змінну starship_collided. Спочатку ми встановимо для цього значення False, і це ще одна умова, за якої основна гра в циклі while завершиться:

```
is_running = True
starship_collided = False
# Main game playing Loop
while is_running and not starship_collided:
```

Таким чином, цикл гри завершиться, якщо користувач вирішить вийти або зіткнеться зоряний корабель з метеором.

Визначення виграшу

Наразі у нас є спосіб програти гру, але у нас немає способу виграти гру! Проте ми хочемо, щоб гравець міг виграти гру, вижившим протягом певного періоду часу. Ми могли б представити це за допомогою якогось таймера. Однак у нашому випадку ми представимо виграш у вигляді певної кількості циклів основного ігрового циклу. Якщо гравець вижив протягом такої кількості циклів, то він виграв. Наприклад:

```
# See if the player has won
if cycle_count == MAX_NUMBER_OF_CYCLES:
    print('WINNER!')
    break
```

У цьому випадку роздруковується повідомлення про те, що гравець виграв, а потім основний ігровий цикл припиняється (за допомогою оператора break).

Глобальне значення MAX_NUMBER_OF_CYCLES можна встановити відповідним чином, наприклад:

```
MAX_NUMBER_OF_CYCLES = 1000
```

Збільшення кількості метеорів

Ми могли б залишити гру як є на даний момент, оскільки тепер можна виграти або програти гру. Однак є кілька речей, які можна легко додати, які покращать враження від гри. Одним з них є збільшення кількості метеорів на екрані, що ускладнює процес у ході гри.

Ми можемо зробити це за допомогою NEW_METEOR_CYCLE_INTERVAL.

```
NEW_METEOR_CYCLE_INTERVAL = 40
```

Коли цей інтервал буде досягнутий, ми можемо додати новий метеор до списку поточних метеорів; потім він буде автоматично відображений класом Game. Наприклад:

```
# Determine if new meteors should be added
if cycle_count % NEW_METEOR_CYCLE_INTERVAL == 0:
    self.meteors.append(Meteor(self))
```

Тепер кожен NEW_METEOR_CYCLE_INTERVAL ще один метеор буде додаватися до гри за випадковою координатою x.

Призупинення гри

Ще одна особливість багатьох ігор - це можливість призупинити гру. Це можна легко додати, відстежуючи клавішу паузи (це може бути буква p, представлена event_key pygame.K_p). Якщо натиснути цю кнопку, гру можна призупинити до повторного натискання клавіші.

Операцію паузи можна реалізувати як метод _pause(), який буде поглинати всі події, доки не буде натиснута відповідна клавіша. Наприклад:

```
def _pause(self):
    paused = True
    while paused:
        for event in pygame.event.get():
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_p:
                    paused = False
                    break
```

У цьому методі зовнішній цикл while буде зациклюватися до тих пір, поки локальна змінна paused не стане False. Це відбувається лише при натисканні клавіші «p». Оператор break після установки змінної paused в False гарантує, що внутрішній цикл for закінчиться, дозволяючи зовнішньому циклу while перевірити значення paused та завершитись.

Метод _pause() можна викликати під час ігрового циклу, відстежуючи клавішу «p» у циклі для події та викликаючи звідти метод _pause():

```
elif event.key == pygame.K_p:
    self._pause()
```

Відображення повідомлення про закінчення гри

PyGame не має простого способу створення спливаючого діалогового вікна для відображення таких повідомлень, як «Ви виграли»; або «Ви програли», тому досі ми використовували оператори print. Однак ми можемо використовувати для цього бібліотеку графічного інтерфейсу, наприклад wxPython, або відобразити повідомлення на поверхні дисплея, щоб вказати, виграв гравець чи програв.

Ми можемо відобразити повідомлення на поверхні дисплея за допомогою класу pygame.font.Font. Тому додамо метод _display_message() до класу Game, який можна використовувати для відображення відповідних повідомлень:

```
def _display_message(self, message):
    """ Displays a message to the user on the screen """
    text_font = pygame.font.Font('freesansbold.ttf', 48)
    text_surface=text_font.render(message, True, BLUE, WHITE)
    text_rectangle = text_surface.get_rect()
    text_rectangle.center=(DISPLAY_WIDTH/2, DISPLAY_HEIGHT/2)
```

```
        self.display_surface.fill(WHITE)
        self.display_surface.blit(text_surface, text_rectangle)
```

Тут провідне нижнє підкреслення в назві методу вказує, що його не слід викликати з-за меж класу Game.

Тепер ми можемо змінити основний цикл так, щоб користувачеві відображалися відповідні повідомлення, наприклад:

```
# Check to see if a meteor has hit the ship
if self._check_for_collision():
    starship_collided = True
    self._display_message('Collision: Game Over')
```

Результат виконання наведеного вище коду під час зіткнення показаний нижче:

Рис. 31 Повідомлення про закінчення гри

Гра StarshipMeteors

Нижче наведено повний листинг остаточної версії гри StarshipMeteors [16]:

```
import pygame, random, time

FRAME_REFRESH_RATE = 30
DISPLAY_WIDTH = 600
DISPLAY_HEIGHT = 400
BLUE = (0, 0, 255)
WHITE = (255, 255, 255)
BACKGROUND = (0, 0, 0)
INITIAL_METEOR_Y_LOCATION = 10
INITIAL_NUMBER_OF_METEORS = 8
MAX_METEOR_SPEED = 5
STARSHIP_SPEED = 10
MAX_NUMBER_OF_CYCLES = 1000
NEW_METEOR_CYCLE_INTERVAL = 40

class GameObject:
    def __init__(self):
        self.x = 0
        self.y = 0
        self.image = None
        self.width = 0
        self.height = 0
    def load_image(self, filename):
        self.image = pygame.image.load(filename).convert()
        self.width = self.image.get_width()
        self.height = self.image.get_height()
    def rect(self):
        """ Generates a rectangle representing the objects location
```

```
        and dimensions """
        return pygame.Rect(self.x, self.y, self.width, self.height)
    def draw(self):
        """ draw the game object at the
            current x, y coordinates """
        self.game.display_surface.blit(self.image, (self.x, self.y))

class Starship(GameObject):
    """ Represents a starship"""
    def __init__(self, game):
        super().__init__()
        self.game = game
        self.x = DISPLAY_WIDTH / 2
        self.y = DISPLAY_HEIGHT - 40
        self.load_image('starship.png')
    def move_right(self):
        """ moves the starship right across the screen """
        self.x = self.x + STARSHIP_SPEED
        if self.x + self.width > DISPLAY_WIDTH:
            self.x = DISPLAY_WIDTH - self.width
    def move_left(self):
        """ Move the starship left across the screen """
        self.x = self.x - STARSHIP_SPEED
        if self.x < 0:
            self.x = 0

    def move_up(self):
        """ Move the starship up the screen """
        self.y = self.y - STARSHIP_SPEED
        if self.y < 0:
            self.y = 0
    def move_down(self):
        """ Move the starship down the screen """
        self.y = self.y + STARSHIP_SPEED
        if self.y + self.height > DISPLAY_HEIGHT:
            self.y = DISPLAY_HEIGHT - self.height
    def __str__(self):
        return 'Starship(' + str(self.x) + ', ' + str(self.y) + ')'

class Meteor(GameObject):
    """ represents a meteor in the game """
    def __init__(self, game):
        super().__init__()
        self.game = game
        self.x = random.randint(0, DISPLAY_WIDTH)
        self.y = INITIAL_METEOR_Y_LOCATION
        self.speed = random.randint(1, MAX_METEOR_SPEED)
        self.load_image('meteor.png')
    def move_down(self):
        """ Move the meteor down the screen """
        self.y = self.y + self.speed
        if self.y > DISPLAY_HEIGHT:
            self.y = 5
    def __str__(self):
        return 'Meteor(' + str(self.x) + ', ' + str(self.y) + ')'

class Game:
    """ Represents the game itself, holds the main game playing
        loop """
```

```
        def __init__(self):
            pygame.init()
            # Set up the display
            self.display_surface =
pygame.display.set_mode((DISPLAY_WIDTH, DISPLAY_HEIGHT))
            pygame.display.set_caption('Starship Meteors')
            # Used for timing within the program.
            self.clock = pygame.time.Clock()
            # Set up the starship
            self.starship = Starship(self)
            # Set up meteors
            self.meteors = [Meteor(self) for _ in range(0,
INITIAL_NUMBER_OF_METEORS)]
        def _check_for_collision(self):
            """ Checks to see if any of the meteors have collided with
                the starship """
            result = False
            for meteor in self.meteors:
                if self.starship.rect().colliderect(meteor.rect()):
                    result = True
                    break
            return result
        def _display_message(self, message):
            """ Displays a message to the user on the screen """
            text_font = pygame.font.Font('freesansbold.ttf', 48)
            text_surface = text_font.render(message, True, BLUE, WHITE)
            text_rectangle = text_surface.get_rect()
            text_rectangle.center = (DISPLAY_WIDTH/2, DISPLAY_HEIGHT/2)
            self.display_surface.fill(WHITE)
            self.display_surface.blit(text_surface, text_rectangle)
        def _pause(self):
            paused = True
            while paused:
                for event in pygame.event.get():
                    if event.type == pygame.KEYDOWN:
                        if event.key == pygame.K_p:
                            paused = False
                            break
        def play(self):
            is_running = True
            starship_collided = False
            cycle_count = 0

            # Main game playing Loop
            while is_running and not starship_collided:
             # Indicates how many times the main game loop has been run
                cycle_count += 1

                # See if the player has won
                if cycle_count == MAX_NUMBER_OF_CYCLES:
                    self._display_message('WINNER!')
                    break

                # Work out what the user wants to do
                for event in pygame.event.get():
                    if event.type == pygame.QUIT:
                        is_running = False
                    elif event.type == pygame.KEYDOWN:
                        # Check to see which key is pressed
```

```
                    if event.key == pygame.K_RIGHT:
                        # Right arrow key has been pressed
                        # move the player right
                        self.starship.move_right()
                    elif event.key == pygame.K_LEFT:
                        # Left arrow has been pressed
                        # move the player left
                        self.starship.move_left()
                    elif event.key == pygame.K_UP:
                        self.starship.move_up()
                    elif event.key == pygame.K_DOWN:
                        self.starship.move_down()
                    elif event.key == pygame.K_p:
                        self._pause()
                    elif event.key == pygame.K_q:
                        is_running = False
            # Move the Meteors
            for meteor in self.meteors:
                meteor.move_down()
            # Clear the screen of current contents
            self.display_surface.fill(BACKGROUND)

            # Draw the meteors and the starship
            self.starship.draw()
            for meteor in self.meteors:
                meteor.draw()

            # Check to see if a meteor has hit the ship
            if self._check_for_collision():
                starship_collided = True
                self._display_message('Collision: Game Over')

            # Determine if new mateors should be added
            if cycle_count % NEW_METEOR_CYCLE_INTERVAL == 0:
                self.meteors.append(Meteor(self))

            # Update the display
            pygame.display.update()

 #Defines the frame rate. The number is number of frames per second
 # Should be called once per frame (but only once)
            self.clock.tick(FRAME_REFRESH_RATE)

        time.sleep(1)
        # Let pygame shutdown gracefully
        pygame.quit()
def main():
    print('Starting Game')
    game = Game()
    game.play()
    print('Game Over')
if __name__ == '__main__':
    main()
```

4.3 Питання для самоконтролю

1. Які засоби мова Python надає для роботи з 2D графікою? Які бібліотеки призначені для роботи з графікою?
2. Яким чином можна відобразити графік математичної функції?
3. Як можна налаштувати колір та тип лінії на графіку математичної функції?
4. Яким чином можна відобразити гістограму?
5. Яким чином можна зберегти зображення у файл?

ЛІТЕРАТУРА

1. Яковенко А.В. Основи програмування. Python. Частина 1: підручник для студ. спеціальності 122 "Комп'ютерні науки". – Київ : КПІ ім. Ігоря Сікорського, 2018. 195 с.
2. Програмування на мові Python. Методичні вказівки до виконання лабораторних робіт студентами денної та заочної форми навчання спеціальностей 123 "Комп'ютерна інженерія", 125 "Кібербезпека" / Укл.: Є.В. Мелешко – Кропивницький: ЦНТУ, 2017. 58 с.
3. Програмування числових методів мовою Python : підруч. / А. В. Анісімов, А. Ю. Дорошенко, С. Д. Погорілий, Я. Ю. Дорогий ; за ред. А. В. Анісімова. – К. : Видавничо-поліграфічний центр "Київський університет", 2014. 640 с.
4. Костюченко А.О. Основи програмування мовою Python: навчальний посібник. – Чернігів: ФОП Баликіна С.М., 2020. 180 с.
5. Креневич А.П. Python у прикладах і задачах. Частина 1. Структурне програмування. Навчальний посібник із дисципліни "Інформатика та програмування" – К.: ВПЦ "Київський Університет", 2017. 206 с.
6. Лутц М. Изучаем Python, том 1, 5-е изд. — СПб.: ООО "Диалектика", 2019. 832 с.
7. Лутц М. Изучаем Python, том 2, 5-е изд. — СПб.: ООО "Диалектика", 2020. 720 с.
8. Joakim Sundnes Introduction to Scientific Programming with Python. – Springer, 2020. 148 p.
9. Bill Lubanovic Introducing Python. – O'Reilly Media, 2020. 597 p.
10. Danjou J. Serious Python: black-belt advice on deployment, scalability, testing, and more. – San Francisco, CA : No Starch Press, Inc., 2019. 258 p.
11. The Python Standard Library – [URL] https://docs.python.org/3.9/library/
12. Johansson R. Numerical Python: Scientific Computing and Data Science Applications with Numpy, SciPy and Matplotlib. – Apress, 2019. 700 p.
13. Christian Hill Learning Scientific Programming with Python – Cambridge University Press, 2020. 557 p.
14. Campesato O. Python 3 and Data Analytics Pocket Primer. – Mercury Learning and information, 2021. 238 p.
15. Sam Morley Applying Math with Python. – Packt Publishing, 2020. 336 p.

16. John Hunt Advanced Guide to Python 3 Programming. – Springer, 2019. 497 p
17. Eric Matthes Python Crash Course. – No Starch Press, 2019. 506 p.
18. Fabrizio Romano, Heinrich Kruger Learn Python Programming Third Edition. – Packt Publishing, 2021. 527 p.