

# Лабораторна робота №4

Жук Дмитро РА-241

**Тема:** Функції, модульність коду, імпорт та повторне використання коду.

**Мета:** Ознайомитися з принципами визначення функцій та методів, організацією імпорту модулів, побудови модульного коду та повторного використання готових рішень у Python. Навчитися застосовувати функції, спадкування, callback-механізми, обробку винятків при імпорті та структуризацію програми на класи для створення стійких і легко розширюваних додатків.

## Хід роботи

### 1 Імпорт (Imports)

Імпорт в Python — це спосіб підключити зовнішні модулі або бібліотеки, щоб не писати все з нуля. Використовується ключове слово `import`, або `from module import something` для конкретних частин. Це сприяє повторному використанню: ти береш готові інструменти від спільноти. У цій програмі імпорти розділені на групи — базові, Tkinter, numpy/matplotlib і serial (з try-except для обробки помилок імпорту). Це робить код адаптивним: якщо serial не доступний, програма переходить на dummy-режим.

Основний фрагмент імпортів:

```
1 import struct
2 import threading
3 import time
4 import random
5 import json
6 import tkinter as tk
7 from tkinter import ttk, messagebox, filedialog
8
```

```

9 import numpy as np
10 import matplotlib.pyplot as plt
11 from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
12
13 try:
14     import serial
15     import serial.tools.list_ports
16     UART_AVAILABLE = True
17 except ImportError:
18     UART_AVAILABLE = False

```

- `import struct` — для розпаковки байтів у float (використовується в unpack).
- `import threading` — для створення потоків (клас `UARTReader` успадковує `threading.Thread`).
- `import tkinter as tk` і `from tkinter import ...` — імпорт GUI-елементів.  
`as tk` — аліас для зручності, щоб не писати повне ім'я кожного разу.
- `import numpy as np` — для масивів даних (`self.data = np.zeros(...)`).
- Try-except навколо `import serial` : якщо бібліотека не встановлена, встановлюється флаг `UART_AVAILABLE = False`, і програма не падає, а йде в симуляцію. Це розумний спосіб імпорту для кросплатформовості.

## 2 Функції (Functions)

Функції в Python — це блоки коду, які виконують конкретне завдання, визначаються через `def name(args):`. Вони приймають аргументи, повертають значення (`return`) і можуть мати дефолтні значення. У цій програмі функції — це методи класів (бо код об'єктно-орієнтований), але є й глобальні (як `if name == "main"`). Вони автоматизують повторювані дії, наприклад, читання даних чи оновлення графіка.

### Функції (методи):

- **Метод `init` у класі `UARTReader`:** Це конструктор, ініціалізує об'єкт. Приймає `callback`, `port` тощо, з дефолтними значеннями (`baudrate=115200`).

```

1 def __init__(self, callback, port=None, baudrate=115200, dummy=False):
2     super().__init__(daemon=True)
3     self.callback = callback
4     self(dummy or not UART_AVAILABLE)
5     self.running = False

```

```
6     self.port = port
7     self.baudrate = baudrate
8     self.ser = None
```

Ця функція встановлює стан потоку. `super().__init__` — викликає конструктор батьківського класу `threading.Thread`. Це приклад спадкування для повторного використання.

- **Метод `run` у `UARTReader`:** Перевизначений метод з `Thread`, запускає логіку потоку. Використовує `if` для вибору режиму.

```
1 def run(self):
2     self.running = True
3     if self.dummy:
4         self._run_dummy()
5     else:
6         try:
7             self.ser = serial.Serial(self.port, self.baudrate, timeout=1)
8             self._run_uart()
9         except Exception as e:
10            print(f"UART open error: {e}")
11            self.running = False
```

Функція не повертає нічого (`None`), але запускає приватні методи (`_run_dummy`, `_run_uart`). Приватні методи (`з _`) — конвенція для "внутрішнього" використання.

- **Метод `on_new_data` у класі `App`:** Callback-функція, яка обробляє нові дані.

```
1 def on_new_data(self, floats):
2     self.data = np.roll(self.data, -1, axis=1)
3     self.data[:, -1] = floats
```

Приймає список `floats`, маніпулює numpy-массивом `textttself.data` (зсув і присвоєння). Це проста функція для оновлення даних, викликається з потоку.

- **Метод `update_plot` у `App`:** Рекурсивно планує себе через `after` для анімації.

```
1 def update_plot(self):
2     if getattr(self, "_closing", False):
3         return
4     if self.config_data and self.lines:
5         for item in self.lines:
6             line, cfg, ch_idx = item
7             scaled = self.data[ch_idx] * cfg.get("scale", 1.0)
```

```

8         line.set_ydata(scaled)
9     for ax in self.subplots:
10        ax.relim()
11        ax.autoscale_view(scaley=True)
12    try:
13        self.canvas.draw()
14    except tk.TclError:
15        return
16    try:
17        self._after_id = self.root.after(100, self.update_plot)
18    except tk.TclError:
19        self._after_id = None

```

Це ключова функція для візуалізації: оновлює лінії, масштабує, перемальовує. Використовує `try-except` для обробки помилок Tkinter.

### 3 Модульність коду (Modularity)

Модульність — це розбиття коду на незалежні модулі (класи, функції, файли), щоб кожна частина робила свою справу і легко замінювалася. У Python це досягається класами, методами і імпортами. Ця програма модульна: логіка читання даних в окремому класі (`UARTReader`), GUI і візуалізація — в `App`.

- **Клас `UARTReader`:** Окремий модуль для читання даних, успадковує `Thread`. Містить методи `run`, `_run_uart`, `_run_dummy`, `stop`. Це ізоляє багатопотокість від GUI.

```

1 class UARTReader(threading.Thread):
2     # ... (all methods inside)

```

Клас — як модуль: створюєш об'єкт, запускаєш, зупиняєш. Модульність: `dummy` режим відділений від реального UART.

- **Клас `App`:** Головний модуль для GUI. Містить `init` для налаштування інтерфейсу, методи для завантаження конфігу, старту/стопу, оновлення.

```

1 class App:
2     def __init__(self, root):
3         # ... menu settings, controls, placeholder graphics
4         # other methods: load_config, setup_plots, refresh_ports, etc.

```

`init` будує інтерфейс (меню, кнопки, canvas). Інші методи — як підмодулі: `setup_plots` очищує і налаштовує графіки, `on_close` чисто закриває програму.

- **Глобальний блок `if name == "main"`:** Це модульність для запуску як скрипту, але дозволяє імпортувати клас `App` в інший файл без запуску.

```
1 if __name__ == "__main__":
2     root = tk.Tk()
3     app = App(root)
4     root.protocol("WM_DELETE_WINDOW", app.on_close)
5     root.mainloop()
```

Це стандартний патерн: програма запускається тільки якщо файл виконується безпосередньо.

## 4 Повторне використання коду (Code Reuse)

Повторне використання — це уникнення дублювання через функції, класи, імпорти і спадкування. У Python це ООП (спадкування), імпорти бібліотек і виклики методів. Тут багато прикладів: готові бібліотеки (`serial`, `matplotlib`), спадкування (`Thread`), виклики методів у циклах.

- **Спадкування від `threading.Thread`:** `UARTReader` повторно використовує логіку потоків з стандартної бібліотеки.

```
1 class UARTReader(threading.Thread):
2     def __init__(self, ...):
3         super().__init__(daemon=True)
4     def run(self): # redefinition
5         ...
```

Замість писати свій потік, беремо готовий і додаємо свою логіку.

- **Виклик `callback` у циклі:** У `_run_uart` і `_run_dummy` повторно використовується `self.callback` для обробки даних.

```
1 # in _run_uart
2 if frame[-2:] == TERMINATOR:
3     floats = struct.unpack('<16f', frame[:64])
4     self.callback(floats)
```

```
1 # in _run_dummy
2 dummy_floats = [random.uniform(0, 10) for _ in range(16)]
```

```
3 self.callback(dummy_floats)
```

Один і той же callback (`on_new_data`) використовується в обох режимах — немає дублювання обробки.

- **Цикли для налаштування в `setup_plots`:** Повторне використання коду в `for` для subplot і каналів.

```
1 for i in range(n_subplots):
2     subplot_cfg = subplots_cfg[i] if i < len(subplots_cfg) else {}
3     ax = self.fig.add_subplot(n_subplots, 1, i+1)
4     # ... settings
5     self.subplots.append(ax)
```

```
1 for ch_idx, ch_name in enumerate(self.channel_keys):
2     # ...
3     self.lines.append((line, ch, ch_idx))
```

Замість писати код для кожного subplot/каналу окремо, цикл повторно використовує логіку для всіх.

- **Імпорти для повторного використання:** NumPy для масивів (`np.roll`, `np.zeros`), Matplotlib для графіків (`ax.plot`, `fig.tight_layout`) — все готове, не пишемо з нуля.

## Висновок

У лабораторній роботі №4 розглянуто організацію Python-програми: імпорт модулів з обробкою винятків, використання функцій і методів класів, принципи модульності (окрім класи `UARTReader` та `App`) та повторне використання коду (спадкування, `callback`, цикли, готові бібліотеки). Програма демонструє стійку багатопоточну візуалізацію даних з UART або в симуляційному режимі.

Звіт оформлено в L<sup>A</sup>T<sub>E</sub>X (`extarticle` 14 pt, XeLaTeX, `listings`, `tcolorbox`), що забезпечило професійний вигляд документу.