COMMENTO CODICE wallet_main.py

```
import os, json, re
from dotenv import load_dotenv
from web3 import Web3
from web3.exceptions import Web3RPCError
from solcx import (
    install_solc, set_solc_version, compile_standard,
    get_installed_solc_versions,
)
```

Questo blocco importa tutte le librerie necessarie per il funzionamento dello script. In particolare, carica le variabili d'ambiente da file `.env`, permette la connessione a un nodo Ethereum tramite Web3, e consente la compilazione del contratto Solidity direttamente da Python usando il compilatore `solcx`.

Un contratto Solidity è un programma eseguibile sulla blockchain di Ethereum che automatizza regole e transazioni. Può gestire fondi, eseguire logiche personalizzate e interagire con altri contratti in modo trasparente, sicuro e immutabile.

```
# ---------- 1. ambiente ----------
load_dotenv()
RPC_URL = os.getenv("WEB3_PROVIDER_URL")
PRIVATE_KEYS = [os.getenv("CHIAVE_PRIVATA_1"), os.getenv("CHIAVE_PRIVATA_2")]
if not RPC_URL or not all(PRIVATE_KEYS):
    raise SystemExit(".env incompleto (URL o chiavi mancanti)")

w3 = Web3(Web3.HTTPProvider(RPC_URL))
if not w3.is_connected():
    raise SystemExit("Nodo Ganache non raggiungibile")

owners      = [w3.eth.account.from_key(k) for k in PRIVATE_KEYS]
owner_addrs = [o.address for o in owners]
print("⛽ block gas limit:", w3.eth.get_block("latest").gasLimit)
```

Questo viene inizializzato l'ambiente leggendo le variabili da un file .env, tra cui l'endpoint del nodo Ethereum (Ganache) e le chiavi private degli account. Verifica la correttezza dei parametri e la connessione al nodo. Se tutto è corretto, crea gli oggetti account (owners) da chiave privata e ne estrae gli indirizzi (owner_addrs) per l'uso nelle transazioni.

Ganache è un ambiente di sviluppo locale per Ethereum che simula una blockchain privata, permettendo di testare smart contract e transazioni in modo rapido e senza costi reali.

```python
# ---------- 2. compilazione ----------
1 usage
def compile_contract(src="wallet_contract.sol", name="wallet_contract"):
    if "0.8.26" not in get_installed_solc_versions():
        install_solc("0.8.26")
    set_solc_version("0.8.26")

    compiled = compile_standard(
        input_data: {
            "language": "Solidity",
            "sources": {src: {"content": open(src).read()}},
            "settings": {"outputSelection": {"*": {"*": ["abi", "evm.bytecode"]}}},
        },
        solc_version="0.8.26",
    )
    abi  = compiled["contracts"][src][name]["abi"]
    byte = compiled["contracts"][src][name]["evm"]["bytecode"]["object"]
    json.dump( obj: {"abi": abi, "bytecode": byte}, open(f"{name}.json", "w"), indent=2)
    print("🛠️  contratto compilato")
    return abi, byte
```

Questa funzione si occupa della compilazione del contratto scritto in Solidity. Controlla se la versione del compilatore `0.8.26` è installata, la imposta e compila il file `wallet_contract.sol`. Estrae l'ABI e il bytecode, salvandoli in un file `.json` per l'uso successivo nella fase di deploy.

ABI e bytecode vengono estratti dal contratto Solidity compilato. In particolare da compiled , che è il risultato di ritorno della compilazione del contratto.
L'ABI (Application Binary Interface) è la descrizione dell'interfaccia del contratto: specifica le funzioni disponibili, i tipi di input/output e gli eventi, permettendo a Web3 o ad altri strumenti di interagire correttamente con il contratto.
Il bytecode è il codice compilato dello smart contract in formato eseguibile dalla Ethereum Virtual Machine (EVM), ed è ciò che viene effettivamente distribuito sulla blockchain.

```python
def send_tx(callable_tx, signer, gas=8_000_000):
    tx = callable_tx.build_transaction({
        "from":    signer.address,
        "nonce":   w3.eth.get_transaction_count(signer.address),
        "gas":     gas,
        "gasPrice": w3.eth.gas_price,
        "chainId":  w3.eth.chain_id,
    })
    signed = w3.eth.account.sign_transaction(tx, signer.key)
    try:
        h = w3.eth.send_raw_transaction(signed.raw_transaction)
        return w3.eth.wait_for_transaction_receipt(h)
    except (ValueError, Web3RPCError) as err:
        raw = err.args[0]
        if isinstance(raw, dict):
            h = raw["data"]["hash"]
        else:
            m = TX_HASH_RE.search(raw)
            if not m:
                raise
            h = m.group(0)
        print("⚠️  invalid opcode ignorato - receipt", h[:10], "…")
        return w3.eth.get_transaction_receipt(h)
```

Questa funzione `send_tx` costruisce, firma e invia una transazione sulla rete Ethereum. Se la transazione genera un errore (`invalid opcode`, ad esempio), cerca di recuperare comunque l'hash per ottenerne il receipt e analizzarla, evitando che il programma si blocchi. Utile per il debug e per garantire maggiore robustezza del flusso.

```python
# ---------- 4. deploy ----------
1 usage
def deploy_wallet(abi, bytecode, addrs, req, deployer):
    factory = w3.eth.contract(abi=abi, bytecode=bytecode)
    rec = send_tx(factory.constructor( *args: addrs, req), deployer)
    print("🚀 wallet @", rec.contractAddress)
    return rec.contractAddress
```

Questa funzione `deploy_wallet` si occupa del deployment del contratto MultiSig sulla blockchain. Usa ABI e bytecode compilati per creare il contratto, lo invia tramite `send_tx`, e stampa l'indirizzo del contratto appena creato. Restituisce infine l'indirizzo utile per interagire col wallet.

```python
# ---------- 5. flow ----------
if __name__ == "__main__":
    abi, bytecode = compile_contract()
    wallet_addr = deploy_wallet(abi, bytecode, owner_addrs, req: 1, owners[0])
    wallet = w3.eth.contract(address=wallet_addr, abi=abi)

    # --- deposito 0.01 ETH ---
    tx_hash = w3.eth.send_transaction(
        {
            "from":  owners[0].address,
            "to":    owners[1].address,
            "value": w3.to_wei( number: 0.01,  unit: "ether"),
            "gas":   210_000,
            "gasPrice": w3.eth.gas_price,
            "nonce": w3.eth.get_transaction_count(owners[0].address),
            "chainId": w3.eth.chain_id,
        }
    )
    w3.eth.wait_for_transaction_receipt(tx_hash)
    print("📥 deposito eseguito")
```

Questa parte del codice rappresenta il flusso principale: compila il contratto, lo distribuisce sulla blockchain e crea un oggetto per interagirci. Successivamente, simula un deposito di 0.01 ETH tra due account, utile per verificare il funzionamento delle transazioni prima di usare il wallet multisig.

```python
    # --- submit ---
    recipient = owner_addrs[1]
    rec = send_tx(
        wallet.functions.submitTransaction(recipient, w3.to_wei( number: 0.002,  unit: "ethe
        owners[0]
    )

    print("✅ transazione eseguita")
```

Questa sezione esegue una transazione multisig usando la funzione `submitTransaction` del contratto: uno degli owner propone un trasferimento di 0.002 ETH a un altro owner. La transazione viene inviata e firmata da `owners[0]`. In uno scenario reale, servirebbero conferme da altri owner per essere eseguita, ma qui è sufficiente una sola firma perché `req=1`.