Full Python Code: Extremely Strong Password Generator

```
"""
Very-hard-to-break deterministic password generator.

Requirements:
    pip install argon2-cffi

Notes:
 - MUST set environment variable LIFETIME_PWD_PEPPER to a long secret stored in a secure vault.
 - Strongly recommended: generate and securely store a per-user high-entropy 'user_secret' (32+ bytes).
 - Tune Argon2 params for your deployment after benchmarking.
"""

import os
import hashlib
import hmac
import base64
import math
import secrets
from datetime import datetime
from typing import Any, Dict, List, Optional

try:
    from argon2.low_level import hash_secret_raw, Type as Argon2Type
except Exception:
    raise ImportError("argon2-cffi required. Run: pip install argon2-cffi")

# ---------------------------
# configuration (tune on deploy)
# ---------------------------
DEFAULT_ARGON2_TIME = 6                 # higher -> slower/stronger
DEFAULT_ARGON2_MEMORY_KIB = 262144      # 256 MiB (increase if you can)
DEFAULT_ARGON2_PARALLELISM = 4
DEFAULT_ARGON2_HASHLEN = 128            # large internal key
DEFAULT_PASSWORD_LENGTH = 48            # long password
# character pool: reasonably safe ASCII printable set
CHAR_POOL = (
    "abcdefghijklmnopqrstuvwxyz"
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    "0123456789"
    "!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~"  # many printable punctuation
)

# ---------------------------
# Helpers: normalize and flatten
# ---------------------------
def normalize_value(v: Any) -> str:
    if v is None:
        return "None"
    if isinstance(v, (int, float, bool)):
        return str(v)
    if isinstance(v, datetime):
        return v.isoformat()
    if isinstance(v, (list, tuple, set)):
        items = sorted(normalize_value(x) for x in v)
        return "[" + ",".join(items) + "]"
    if isinstance(v, dict):
        items = sorted((str(k), normalize_value(vv)) for k, vv in v.items())
        return "{" + ",".join(f"{k}:{vv}" for k, vv in items) + "}"
    return str(v)

def collect_and_flatten(data: Dict[str, Any]) -> bytes:
    parts: List[str] = []
    for key in sorted(data.keys()):
        parts.append(f"{key}={normalize_value(data[key])}")
    joined = "|".join(parts)
```

```
    return joined.encode("utf-8")

# ---------------------------
# Pre-hash and combine secrets
# ---------------------------
def pre_hash_master(master_bytes: bytes) -> bytes:
    # SHA3-512 prehash for structure hiding
    return hashlib.sha3_512(master_bytes).digest()

def combine_with_user_secret_and_pepper(prehashed: bytes, user_secret: bytes, pepper: bytes) -> bytes:
    """
    Combine deterministically and securely:
    HMAC-SHA512 keyed by pepper over (prehashed || user_secret)
    """
    return hmac.new(pepper, prehashed + user_secret, hashlib.sha512).digest()

# ---------------------------
# Argon2 derivation
# ---------------------------
def derive_argon2(master_key: bytes,
                  salt: bytes,
                  time_cost: int,
                  memory_cost_kib: int,
                  parallelism: int,
                  hash_len: int) -> bytes:
    # Argon2id raw output
    return hash_secret_raw(secret=master_key,
                           salt=salt,
                           time_cost=time_cost,
                           memory_cost=memory_cost_kib,
                           parallelism=parallelism,
                           hash_len=hash_len,
                           type=Argon2Type.ID)

# ---------------------------
# Expand and map to password
# ---------------------------
def hkdf_expand(key_material: bytes, info: bytes, length: int) -> bytes:
    # Simple HKDF-SHA256 expand
    prk = hashlib.sha256(key_material).digest()
    okm = b""
    previous = b""
    i = 1
    while len(okm) < length:
        digest = hmac.new(prk, previous + info + bytes([i]), hashlib.sha256).digest()
        okm += digest
        previous = digest
        i += 1
    return okm[:length]

def bytes_to_password(bytes_seq: bytes, length: int, require_classes: bool = True) -> str:
    pool = CHAR_POOL
    out_chars: List[str] = []
    pos = 0
    classes = [
        "abcdefghijklmnopqrstuvwxyz",
        "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
        "0123456789",
        "!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~"
    ] if require_classes else []

    def pick(bslice: bytes, chars: str) -> str:
        num = int.from_bytes(bslice, "big")
        return chars[num % len(chars)]

    for cls in classes:
        b = bytes_seq[pos:pos+4] if pos+4 <= len(bytes_seq) else (bytes_seq[pos:] + bytes_seq[:4 - (len(
        out_chars.append(pick(b, cls))
        pos += 4

    while len(out_chars) < length:
        b = bytes_seq[pos:pos+4] if pos+4 <= len(bytes_seq) else (bytes_seq[pos:] + bytes_seq[:4 - (len(
```

```
        out_chars.append(pick(b, pool))
        pos += 4

    order_key = hmac.new(bytes_seq, "".join(out_chars).encode("utf-8"), hashlib.sha256).digest()
    indices = list(range(len(out_chars)))
    j = 0
    for i in range(len(indices)-1, 0, -1):
        swap_with = order_key[j % len(order_key)] % (i+1)
        indices[i], indices[swap_with] = indices[swap_with], indices[i]
        j += 1
    shuffled = [out_chars[i] for i in indices]
    return "".join(shuffled)

# ---------------------------
# High-level API
# ---------------------------
def generate_extremely_strong_password(user_data: Dict[str, Any],
                                       salt_identifier: str,
                                       user_secret: Optional[bytes] = None,
                                       password_length: int = DEFAULT_PASSWORD_LENGTH,
                                       time_cost: int = DEFAULT_ARGON2_TIME,
                                       memory_cost_kib: int = DEFAULT_ARGON2_MEMORY_KIB,
                                       parallelism: int = DEFAULT_ARGON2_PARALLELISM,
                                       hash_len: int = DEFAULT_ARGON2_HASHLEN) -> Dict[str, Any]:

    pepper_env = os.environ.get("LIFETIME_PWD_PEPPER")
    if not pepper_env:
        raise RuntimeError("PEPPER not set. Set LIFETIME_PWD_PEPPER env var to a long secret stored in v
    pepper = pepper_env.encode("utf-8")

    created_secret = False
    if user_secret is None:
        user_secret = secrets.token_bytes(32)
        created_secret = True

    master_bytes = collect_and_flatten(user_data)
    prehashed = pre_hash_master(master_bytes)
    combined = combine_with_user_secret_and_pepper(prehashed, user_secret, pepper)
    salt = hashlib.sha256(salt_identifier.encode("utf-8")).digest()

    arg_key = derive_argon2(master_key=combined,
                            salt=salt,
                            time_cost=time_cost,
                            memory_cost_kib=memory_cost_kib,
                            parallelism=parallelism,
                            hash_len=hash_len)

    need_bytes = max(password_length * 4, 128)
    expanded = hkdf_expand(arg_key, info=b"lifetime-password-v1", length=need_bytes)
    pwd = bytes_to_password(expanded, password_length, require_classes=True)
    entropy_est = estimate_entropy(pwd := pwd)

    result = {
        "password": pwd,
        "entropy_bits_estimate": entropy_est,
        "salt_b64": base64.b64encode(salt).decode("utf-8"),
        "argon2_params": {
            "time_cost": time_cost,
            "memory_cost_kib": memory_cost_kib,
            "parallelism": parallelism,
            "hash_len": hash_len
        },
        "password_length": password_length,
        "note": "Store user_secret (if newly generated) securely. PEPPER must be secret."
    }
    if created_secret:
        result["generated_user_secret_b64"] = base64.b64encode(user_secret).decode("utf-8")
    return result

# ---------------------------
# Simple entropy estimator
# ---------------------------
```

```
def estimate_entropy(password: str) -> float:
    pool = 0
    if any(c.islower() for c in password): pool += 26
    if any(c.isupper() for c in password): pool += 26
    if any(c.isdigit() for c in password): pool += 10
    if any(c in "!\"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~" for c in password): pool += 32
    if pool == 0:
        return 0.0
    return math.log2(pool) * len(password)

# ----------------------------
# Example usage
# ----------------------------
if __name__ == "__main__":
    example = {
        "birthdate": "1987-05-14",
        "important_dates": ["2005-06-20", "2010-09-01", "2019-11-02"],
        "favorite_numbers": [3, 7, 42],
        "height_cm": 178,
        "weight_kg": 75,
        "phone_last4": "4321",
    }
    out = generate_extremely_strong_password(example, salt_identifier="user@example.com", user_secret=No
                                             password_length=48,
                                             time_cost=6,
                                             memory_cost_kib=262144,
                                             parallelism=4,
                                             hash_len=128)
    print("Password:", out["password"])
    print("Entropy estimate (bits):", round(out["entropy_bits_estimate"], 2))
    if "generated_user_secret_b64" in out:
        print("Generated user_secret:", out["generated_user_secret_b64"])
    print("Argon2 params:", out["argon2_params"])
    print(out["note"])
```