```
from firedrake import *
import numpy as np

# Mesh - use a built in meshing function:
mesh = UnitSquareMesh(40, 40, quadrilateral=True)
left, right, bottom, top = 1, 2, 3, 4  # Boundary IDs
n = FacetNormal(mesh)  # Normals, required for Nusselt number
domain_volume = assemble(1.*dx(domain=mesh))  # Required for RMS velocity

# Function spaces:
V = VectorFunctionSpace(mesh, family="CG", degree=2)  # Velocity function space (vector)
W = FunctionSpace(mesh, family="CG", degree=1)  # Pressure function space (scalar)
Q = FunctionSpace(mesh, family="CG", degree=2)  # Temperature function space (scalar)
Z = MixedFunctionSpace([V, W])  # Mixed function space

# Test functions and functions to hold solutions:
v, w = TestFunctions(Z)
q = TestFunction(Q)
z = Function(Z)
u, p = split(z)  # Returns symbolic UFL expression for u and p
Told, Tnew = Function(Q, name="OldTemp"), Function(Q, name="NewTemp")
Ttheta = 0.5 * Tnew + 0.5 * Told  # Temporal discretisation through Crank-Nicholson

# Initialise temperature field:
X = SpatialCoordinate(mesh)
Told.interpolate(1.0 - X[1] + 0.05 * cos(pi * X[0]) * sin(pi * X[1]))
Tnew.assign(Told)

# Important constants:
Ra, mu, kappa, delta_t = Constant(1e4), Constant(1.0), Constant(1.0), Constant(1e-6)
k = Constant((0, 1))  # Unit vector (in direction opposite to gravity)

# Stokes equations in UFL form:
stress = 2 * mu * sym(grad(u))
F_stokes = inner(grad(v), stress) * dx - div(v) * p * dx - (dot(v, k) * Ra * Ttheta) * dx
F_stokes += -w * div(u) * dx  # Continuity equation
# Energy equation in UFL form:
F_energy = q * (Tnew - Told) / delta_t * dx + q * dot(u, grad(Ttheta)) * dx + dot(grad(q), kappa * grad(Ttheta))
     * dx

# Set up boundary conditions and deal with nullspaces:
bcvx, bcvy = DirichletBC(Z.sub(0).sub(0), 0, sub_domain=(left, right)), DirichletBC(Z.sub(0).sub(1), 0,
     sub_domain=(bottom, top))
bctb, bctt = DirichletBC(Q, 1.0, sub_domain=bottom), DirichletBC(Q, 0.0, sub_domain=top)
p_nullspace = MixedVectorSpaceBasis(Z, [Z.sub(0), VectorSpaceBasis(constant=True)])

# Initialise output:
output_file = File('output.pvd')  # Create output file
u_, p_ = z.split()
u_.rename("Velocity"), p_.rename("Pressure")

# Solver dictionary:
solver_parameters = {
    "mat_type": "aij",
    "snes_type": "ksponly",
    "ksp_type": "preonly",
    "pc_type": "lu",
    "pc_factor_mat_solver_type": "mumps"}

# Setup problem and solver objects so we can reuse (cache) solver setup
stokes_problem = NonlinearVariationalProblem(F_stokes, z, bcs=[bcvx, bcvy])
stokes_solver = NonlinearVariationalSolver(stokes_problem, solver_parameters=solver_parameters, nullspace=
     p_nullspace, transpose_nullspace=p_nullspace)
energy_problem = NonlinearVariationalProblem(F_energy, Tnew, bcs=[bctb, bctt])
energy_solver = NonlinearVariationalSolver(energy_problem, solver_parameters=solver_parameters)

# Timestepping aspects
no_timesteps, target_cfl_no = 2000, 1.0
ref_u = Function(V, name="Reference_Velocity")


def compute_timestep(u):
    """Return the timestep, using CFL criterion"""
    tstep = (1. / np.abs(ref_u.interpolate(dot(JacobianInverse(mesh), u)).dat.data).max()) * target_cfl_no
    return tstep


for timestep in range(0, no_timesteps):
    if timestep > 0:
        delta_t.assign(compute_timestep(u))
    if timestep % 10 == 0:
        output_file.write(u_, p_, Tnew)
    stokes_solver.solve()
    energy_solver.solve()
    vrms = sqrt(assemble(dot(u, u) * dx)) * sqrt(1./domain_volume)
    nu_top = -1. * assemble(dot(grad(Tnew), n) * ds(top))
    Told.assign(Tnew)
```