Below is a conceptual outline of how you might structure your Python code and data for a time-dependent traveling algorithm that considers not just distances but also “footfall” (how busy each location is at different hours) and operating hours. This is only one possible approach—there are many ways to tackle this problem, and the actual implementation details will vary depending on your exact requirements and the optimization techniques you choose.

---

1. Data Structures

1. **Place Information**
   A convenient way is to create a class (or a namedtuple / dataclass) that stores each place’s relevant information:
   - **Name** (string)
   - **Opening time** and **closing time** (could be stored as integers for hours, or as `datetime.time` objects)
   - **Footfall profile** (a dictionary mapping an hour of the day -> expected footfall, or a list indexed by hour)
   - (Optional) If each place has unique attributes, like categories or must-see attractions, store them here as well.

python
```
class Place:
    def __init__(self, name, open_time, close_time, footfall_by_hour):
        """
        open_time, close_time: could be integers (e.g., 9 for 9 AM, 20
for 8 PM)
        footfall_by_hour: dict or list that tells us how busy it is at
each hour
        """
        self.name = name
        self.open_time = open_time
        self.close_time = close_time
        self.footfall_by_hour = footfall_by_hour

    def is_open_at(self, hour):
        return self.open_time <= hour < self.close_time
```

2. **Travel Times**
   For the travel-time data between places, you need it to be time-dependent. You can store it in:

    - A **2D dictionary** (or nested dictionary) such that `travel_time[(placeA, placeB)][hour]` returns the travel time from `placeA` to `placeB` if departing at `hour`.
    - Alternatively, a **matrix** (list of lists) indexed by `[placeA_id][placeB_id]`, containing either a function or a table of travel times by hour.

python
```
# Example using a dictionary of dictionaries for time-dependent travel times:
travel_time = {
    ('PlaceA', 'PlaceB'): {
        8: 15,  # 15 minutes if you depart at 08:00
        9: 20,  # 20 minutes if you depart at 09:00
        # ...
    },
    ('PlaceB', 'PlaceA'): {
        8: 10,  # sometimes different from PlaceA -> PlaceB
        9: 12,
        # ...
    },
    # ...
}
```

3. **Graph Representation**
    - Conceptually, each place is a node in a graph.
    - The edges carry not just a single distance or cost, but a function/table that depends on departure time.
    - You could wrap this logic in a class (e.g., `TimeDependentGraph`) that provides a function to query travel times.

---

2. Core Idea of the Algorithm

You want to find an order (a route) to visit a subset or all of the places while minimizing:

1. **Total travel time**
2. **Visiting time at peak footfall** (you want to avoid high footfall if possible)
3. **Respecting opening/closing times**

There is no single “correct” approach. TSP-like problems are NP-hard, so for large numbers of places, you might resort to heuristics, metaheuristics, or approximations (e.g., genetic algorithms, simulated annealing, or branch-and-bound with pruning).

Possible Approaches

1. **Exact (Small Number of Places)**
   - Use a dynamic programming approach (Held-Karp algorithm) but adapt it to time-dependent edges and additional constraints (opening hours, footfall-based penalties).
   - This can get very complex quickly, but it’s do-able for a small set of places.
2. **Heuristic / Metaheuristic (Medium or Large Number of Places)**
   - Use a **Genetic Algorithm**, **Simulated Annealing**, or **Ant Colony Optimization** to get a good (not necessarily optimal) route.
   - Incorporate time-based constraints by recalculating the cost of traveling each edge based on the time you arrive.
3. **Greedy + Local Search**
   - Start with a naive route (e.g., sorted by distance or footfall).
   - Improve it step by step by swapping adjacent places in the route, or by other local search methods, evaluating any improvement in the objective function.

---

3. Storing and Processing the Data

Step-by-Step Outline

1. **Load / Initialize Data**
   - Create a list or dictionary of `Place` objects.
   - Create a structure for travel times (either a dictionary of dictionaries or a function that calculates it on the fly).

python
```python
# Example: Creating place objects
places = [
    Place(
        name="PlaceA",
        open_time=9,
        close_time=20,
        footfall_by_hour={
            8: 0.5, 9: 0.7, 10: 0.8, 11: 0.9, # ...
        }
    ),
    Place(
        name="PlaceB",
        open_time=10,
        close_time=22,
        footfall_by_hour={
            9: 0.3, 10: 0.5, 11: 0.6, # ...
        }
    ),
    # ...
]
# A quick way to access them by name if needed:
place_dict = {place.name: place for place in places}
```

2. **Define a Function for "Cost" of Visiting at a Given Hour**
    - This function can combine footfall cost and any penalties for arriving during closed hours.
    - For instance, you might add a large penalty if you arrive when the place is closed and you have to wait.

python
```
def visit_cost(place, arrival_hour):
    """
    Combine footfall + waiting time if closed
    """
    if not place.is_open_at(arrival_hour):
        # Suppose we wait until it opens
        if arrival_hour < place.open_time:
            waiting = place.open_time - arrival_hour
            # big penalty for waiting
            cost = 10 * waiting
        else:
            # can't visit if it closes before arrival_hour
            # add huge penalty to discourage this route
            cost = 999999
    else:
        # A function of footfall, e.g. multiply by some factor
        cost = place.footfall_by_hour.get(arrival_hour, 1.0)
    return cost
```

3. **Define a Function to Get Time-Dependent Travel Time**
    - Could be as simple as looking up from your dictionary if your travel-time data is precomputed.
    - Or you could implement a function that approximates it based on typical traffic patterns.

python
```
def get_travel_time(origin, destination, departure_hour,
travel_time_data):
    # Adjust departure_hour as needed, e.g., convert to 24-hour format
if partial hours
    return travel_time_data.get((origin, destination),
{}).get(departure_hour, 999999)
```

4. **Construct the Route (High-Level Pseudocode)**
    - Let's say you use a simple backtracking or a dynamic programming approach for a small set of places:

python
```
from functools import lru_cache

# We create an index for each place
index_map = {p.name: i for i, p in enumerate(places)}
N = len(places)

@lru_cache(None)
def best_route(current_place_idx, visited_mask, current_hour):
    """
    current_place_idx: index of the current place
    visited_mask: bitmask representing which places have been visited
    current_hour: the hour at which we arrive at current_place_idx
    """
    if visited_mask == (1 << N) - 1:
        # visited all
        return 0  # or return cost to get back to start, if needed

    best_cost = float('inf')
    for next_place_idx in range(N):
        if not (visited_mask & (1 << next_place_idx)):
            # place not visited yet
            place_name_current = places[current_place_idx].name
            place_name_next = places[next_place_idx].name

            # Travel time from current to next
            t_time = get_travel_time(place_name_current,
place_name_next, current_hour, travel_time)
            arrival_time_next = (current_hour + t_time) % 24  # if
crossing midnight, mod 24

            # Cost of visiting next place
            visit_c = visit_cost(places[next_place_idx],
arrival_time_next)

            # total cost = travel cost + waiting + footfall cost
            total_cost_for_edge = t_time + visit_c

            # Recur
            new_mask = visited_mask | (1 << next_place_idx)
            cost_sub = best_route(next_place_idx, new_mask,
arrival_time_next)
            candidate_cost = total_cost_for_edge + cost_sub

            if candidate_cost < best_cost:
                best_cost = candidate_cost

    return best_cost

def solve_tsp_time_dependent():
    # Try starting at each place or fix a starting place
    best_global_cost = float('inf')
    best_start = None
    for start_idx in range(N):
        cost = best_route(start_idx, 1 << start_idx, 9)  # e.g., start
at 9 AM
        if cost < best_global_cost:
            best_global_cost = cost
            best_start = start_idx
    print("Best route cost:", best_global_cost)
    return best_global_cost
```

5. This (very simplified) dynamic programming approach tries all possible sequences and keeps track of the best. However, it may be slow for large NNN (exponential time complexity).
6. **Heuristic / Metaheuristic Approaches**
   - If the number of places is large, an exact approach becomes infeasible.
   - You could store your route as a **list of place indices** and then apply operators like "swap two places," "reverse a segment," etc., to iterate towards a better solution.

python
```python
def calculate_route_cost(route, start_hour):
    total_cost = 0
    current_hour = start_hour
    for i in range(len(route) - 1):
        origin = route[i]
        destination = route[i+1]
        t_time = get_travel_time(origin.name, destination.name,
current_hour, travel_time)
        arrival_hour = (current_hour + t_time) % 24
        # cost for arriving at destination
        v_cost = visit_cost(destination, arrival_hour)
        total_cost += t_time + v_cost
        current_hour = arrival_hour
    return total_cost

# Then you define a function that tries to improve the route by local
changes.
```

---

4. Putting It All Together

In summary:

1. **Create data structures** for places and time-dependent travel times.
2. **Define utility functions**:
   - `is_open_at(hour)`: check if a place is open.
   - `visit_cost(place, hour)`: compute cost of being at a place at a given hour (footfall + waiting penalties).
   - `get_travel_time(origin, destination, hour, travel_time_data)`: retrieve travel time from your time-dependent data.
3. **Define an objective function** that sums up travel-time costs and footfall costs.

4. **Implement a search/optimization method** (DP, branch-and-bound, local search, or a metaheuristic) that tries to find the route that minimizes your objective function.

Your final code will have to pay attention to time windows (opening/closing times), footfall patterns, waiting times, and the rolling arrival time at each place. The data structures above (a list of `Place` objects, a dictionary-of-dictionaries for travel times) and the helper functions serve as the backbone for whichever optimization method you choose.

Extra Considerations

- **Waiting times**: If you arrive too early, you might decide to wait. If you arrive after closing, that path is invalid (or heavily penalized).
- **Partial hours**: If you want a more granular approach (minutes instead of hours), your data structures need to store footfall and travel times in smaller increments. This increases complexity.
- **Real-time or predicted data**: If you want to adapt to real-time traffic changes, you'll need to be able to update the `travel_time` data structure dynamically.

This gives you a high-level roadmap for how the code might look. The key is to clearly separate **data loading** (places, footfall, travel times) from **the logic** that computes route costs and runs the optimization.