Cheatsheet

Useful commands for Python

Variables and Types

Variables

- Definition: Containers for storing information.
- Example: `x = 10`

Data Types

- Integers (int): Whole numbers (e.g., count of dates).
- Floats (float): Decimal numbers (e.g., compatibility score).
- Booleans (bool): True/False values (e.g., availability).
- Strings (str): Text values (e.g., names).

```
name = "Alexander"  # String variable
flags = 0           # Integer variable
butterflies = True  # Boolean variable
```

Type Conversion

- Checking: Use `type()` to check the type of a variable.
- Conversion:
  - `int()`: Converts to integer.
  - `float()`: Converts to float.
  - `str()`: Converts to string.
  - `bool()`: Converts to boolean.

String Formatting

- Concatenation: Combine strings using `+`.
- Formatting: Use `f"..."` for formatted strings.

```
name = "Alexander"
print(f"Hello, {name}!")
```

```
Hello, Alexander!
```

Comparisons

Comparison Operators

Symbol	Meaning	Example
==	Equal to	score == 100
!=	Not equal to	degree != “Computer Science”
<	Less than	salary < 80000
>	Greater than	experience > 5
<=	Less than or equal to	age <= 65
>=	Greater than or equal to	test_score >= 80

Logical Operators

Symbol	Meaning	Example
and	Both conditions must be true	score > 80 and experience > 5
or	At least one condition must be true	score > 80 or experience > 5
not	Condition must be false	not (score > 80)

Decision-Making

if Statements

- Structure:

```
if condition:
    # code to execute if condition is True
```

- Example:

```
flat_rating = 8
if flat_rating >= 7:
    print("This is a good apartment!")
```

```
This is a good apartment!
```

if-else Statements

- Structure:

```python
if condition:
    # code to execute if condition is True
else:
    # code to execute if condition is False
```

- Example:

```python
flat_rating = 4
if flat_rating >= 7:
    print("Apply for this flat!")
else:
    print("Keep searching!")
```

```
Keep searching!
```

if-elif-else Statements

- Structure:

```python
if condition:
    # code to execute if condition is True
elif condition:
    # code to execute if condition is False
else:
    # code to execute if condition is False
```

- Example:

```python
flat_rating = 8
if flat_rating >= 9:
    print("Amazing flat - apply immediately!")
elif flat_rating >= 7:
    print("Good flat - consider applying")
else:
    print("Keep looking")
```

```
Good flat - consider applying
```

Complex Conditions

- Nested if Statements: Use if statements inside other if statements.
- Logical Operators: Combine conditions using `and`, `or`, `not`.
- Structure:

```python
if (condition1) and (condition2):
    # code if both conditions are True
elif (condition1) or (condition2):
    # code if at least one condition is True
```

```
else:
    # code if none of the conditions are True
```

- Example:

```
flat_rating = 9
price = 900
if (flat_rating >= 9) and (price < 1000):
    print("Amazing flat - apply immediately!")
```

```
Amazing flat - apply immediately!
```

Lists and Tuples

Lists

- Definition: Ordered, mutable collections of items.
- Creation: Use square brackets `[]`.

```
ratings = [4.5, 3.8, 4.2]
restaurants = ["Magic Place", "Sushi Bar", "Coffee Shop"]
```

Accessing Elements

- Indexing: Use `[index]` to access elements.

```
print(restaurants[0])  # Access the first element
```

```
Magic Place
```

- Negative Indexing: Use `[-1]` to access the last element.

```
print(restaurants[-1])  # Access the last element
```

```
Coffee Shop
```

- Slicing: Use `[start:end]` to access a range of elements.

```
print(restaurants[0:2])  # Access the first two elements
```

```
['Magic Place', 'Sushi Bar']
```

Adding Elements

- Appending: Use `append()` to add an element to the end of the list.

```
restaurants.append("Pasta Place")
```

- Inserting: Use `insert()` to add an element at a specific index.

```
restaurants.insert(0, "Pasta Magic")
```

Removing Elements

- Removing: Use `remove()` to remove an element by value.

```
restaurants.remove("Pasta Place")
```

- Removing by Index: Use `pop()` to remove an element by index.

```
restaurants.pop(0)
```

```
'Pasta Magic'
```

Nested Lists

- Definition: Lists containing other lists or tuples.
- Accessing: Use nested indexing.

```
restaurant_data = [
    ["Pasta Place", 4.5, 3],
    ["Sushi Bar", 4.2, 1]
]
print(restaurants[0][1])  # Access the second element of the first list
```

```
a
```

Tuples

- Definition: Ordered, immutable collections of items.
- Creation: Use parentheses `()`.
- Immutability: Once created, cannot be changed.
- Memory Efficiency: Use less memory than lists.
- Use Cases: Ideal for fixed data (e.g., restaurant location).

```
ratings = (4.5, 3.8, 4.2)
restaurant_info = ("Pasta Place", "Italian", 2020)
```

Loops

for Loops

- Definition: Iterate over a sequence of items.
- Structure:

```
for item in sequence:
    # code to execute for each item
```

- Example:

```
treatments = ["Standard Drug", "New Drug A", "New Drug B"]
for treatment in treatments:
    print(f"Evaluating efficacy of {treatment}")
```

```
Evaluating efficacy of Standard Drug
Evaluating efficacy of New Drug A
Evaluating efficacy of New Drug B
```

Range in for Loops

- Definition: Generate a sequence of numbers.
- Structure:

```
range(start, stop, step)
```

- Example:

```
for phase in range(5):  # 0 to 4
    print(f"Starting Phase {phase + 1}")
```

```
Starting Phase 1
Starting Phase 2
Starting Phase 3
Starting Phase 4
Starting Phase 5
```

```
for phase in range(1, 5):  # 1 to 4
    print(f"Starting Phase {phase}")
```

```
Starting Phase 1
Starting Phase 2
Starting Phase 3
Starting Phase 4
```

```python
for phase in range(1, 5, 2):  # 1 to 4, step 2
    print(f"Starting Phase {phase}")
```

```
Starting Phase 1
Starting Phase 3
```

break and continue

- break: Exit the loop.
- continue: Skip the current iteration and continue with the next.

```python
efficacy_scores = [45, 60, 75, 85, 90]
for score in efficacy_scores:
    if score < 50:
        continue
        print(f"Treatment efficacy: {score}%")
    if score >= 85:
        break
```

Tuple unpacking

- Definition: Assign elements of a tuple to variables.
- Structure:
- Example:

```python
restaurant_info = ("Pasta Place", "Italian", 2020)
name, cuisine, year = restaurant_info
print(name)
print(cuisine)
print(year)
```

```
Pasta Place
Italian
2020
```

while Loops

- Definition: Execute code repeatedly as long as a condition is true.
- Structure:

```python
while condition:
    # code to execute while condition is True
```

- Example:

```python
phase = 1
while phase <= 5:
    print(f"Starting Phase {phase}")
    phase += 1
```

```
Starting Phase 1
Starting Phase 2
Starting Phase 3
Starting Phase 4
Starting Phase 5
```

Functions

Basic Function

- Definition: Use the `def` keyword.
- Structure:

```python
def function_name(parameters):
    # code to execute (function body)
    return value  # Optional
```

- Example:

```python
def greet_visitor(name):
    return f"Welcome to the library, {name}!"

greet_visitor("Student")
```

```
'Welcome to the library, Student!'
```

Return Value

- Definition: The value returned by a function.
- Example:

```python
def multiply_by_two(number):
    return number * 2

result = multiply_by_two(5)
print(result)
```

```
10
```

- Note: If a function does not return a value, it implicitly returns `None`.

Default Parameters

- Definition: Provide default values for function parameters.
- Structure:

```
def greet_visitor(name="People"):
    return f"Welcome to the library, {name}!"

print(greet_visitor()) # Calls the function with the default parameter
print(greet_visitor("Tobias")) # Calls the function with a custom parameter
```

Multiple Parameters

- Definition: Functions can have multiple parameters.
- Structure:

```
def greet_visitor(name, age):
    return f"Welcome to the library, {name}! You are {age} years old."

print(greet_visitor("Tobias", 30))
```

String Methods

- Definition: Methods are functions that are called on strings.
- Structure:

```
string.method()
```

- Common String Methods:
  - `.strip()` - Removes whitespace from start and end
  - `.title()` - Capitalizes first letter of each word
  - `.lower()` - Converts to lowercase
  - `.upper()` - Converts to uppercase
- Example:

```
title = "the hitchhikers guide"
print(title.title())
```

```
The Hitchhikers Guide
```

```
title = "   the hitchhikers guide    "
print(title.strip())
```

```
the hitchhikers guide
```

Packages

Standard Libraries

- Definition: Libraries that are part of the Python standard library.
- Access: Import them using `import`.

```python
import math
import random
```

- For long package names, you can use the `as` keyword to create an alias.

```python
import random as rd
```

- To call a function from an imported package, use the package name as a prefix.

```python
random_number = rd.random()
print(random_number)
```

```
0.19401339194908518
```

Installing Packages

- Definition: Install packages using `uv`. Note, don't do this inside of a notebook but in the terminal in your project folder!

```
{bash}
uv add package_name
```

Probability Distributions

Normal Distribution

- When to Use: Most common in business and nature; symmetric outcomes around a mean
- Characteristics:
  - Bell-shaped, symmetric curve
  - Most values cluster around the mean
  - Rare extreme values in tails
- Examples:
  - Investment returns
  - Manufacturing variations
  - Employee performance scores
  - Measurement errors

Python Syntax:

```python
import numpy as np

# Generate normal distribution
returns = np.random.normal(loc=mean, scale=std_dev, size=n_samples)
```

```python
# Example: Stock returns with 10% mean, 15% volatility
stock_returns = np.random.normal(loc=0.10, scale=0.15, size=10000)
```

Parameters:

- `loc`: The mean (center) of the distribution
- `scale`: The standard deviation (spread)
- `size`: Number of samples to generate

Uniform Distribution

- When to Use: Complete uncertainty within a range; all outcomes equally likely
- Characteristics:
  - Flat distribution
  - All values equally likely
  - Hard boundaries (min/max)
  - No clustering around any value
- Examples:
  - Random wait times
  - Initial demand estimates with only min/max known
  - Random sampling from a range

Python Syntax:

```python
# Generate uniform distribution
values = np.random.uniform(low=minimum, high=maximum, size=n_samples)

# Example: Demand between 1000 and 5000 units
demand = np.random.uniform(low=1000, high=5000, size=10000)
```

Parameters:

- `low`: Minimum value (inclusive)
- `high`: Maximum value (exclusive)
- `size`: Number of samples to generate

Exponential Distribution

- When to Use: Time between events; waiting times
- Characteristics:
  - Many small values, few large ones
  - Always positive
  - Memoryless property
  - Right-skewed (long tail)
- Examples:
  - Time between customer arrivals
  - Equipment failure times
  - Time until next sale

    - Duration of phone calls

Python Syntax:

```
# Generate exponential distribution
wait_times = np.random.exponential(scale=average_time, size=n_samples)

# Example: Time between customers (avg 5 minutes)
arrivals = np.random.exponential(scale=5, size=10000)
```

Parameters:

- `scale`: The average (mean) time between events
- `size`: Number of samples to generate

Binomial Distribution

- When to Use: Fixed number of independent yes/no trials
- Characteristics:
    - Discrete outcomes (counts)
    - Fixed number of trials
    - Each trial has same probability
    - Trials are independent
- Examples:
    - Number of defective items in a batch
    - Number of successful sales calls
    - Number of customers who convert
    - Number of loans that default

Python Syntax:

```
# Generate binomial distribution
successes = np.random.binomial(n=n_trials, p=prob_success, size=n_samples)

# Example: 100 sales calls with 20% conversion rate
conversions = np.random.binomial(n=100, p=0.20, size=10000)
```

Parameters:

- `n`: Number of trials
- `p`: Probability of success on each trial
- `size`: Number of experiments to simulate

Common Risk Metrics

Calculate from simulated results:

```
# Basic statistics
mean_return = results.mean()
std_dev = results.std()
```

```python
min_value = results.min()
max_value = results.max()

# Percentiles (Value at Risk)
var_5 = np.percentile(results, 5)  # 5th percentile (worst 5%)
var_95 = np.percentile(results, 95)  # 95th percentile (best 5%)

# Probability of loss
prob_loss = (results < 0).mean()

# Expected shortfall (average of worst 5%)
worst_5_percent = results[results <= var_5]
expected_shortfall = worst_5_percent.mean()

# Correlation between two variables
correlation = np.corrcoef(returns1, returns2)[0, 1]
```

Monte Carlo Simulation

Basic Simulation Pattern

Definition: Running many scenarios to understand possible outcomes under uncertainty.

Common Pattern: 1. Create empty list to store results: `results = []` 2. Run simulations in a loop, calling simulation function 3. Append each result to list: `results.append(simulation_result)` 4. Convert to DataFrame: `pd.DataFrame(results)`

```python
# Example simulation function
def simulate_business_day():
    customers = np.random.normal(100, 20)  # Uncertain demand
    revenue = customers * np.random.uniform(8, 12)  # Variable pricing
    profit = revenue - 500  # Fixed costs
    return {'customers': customers, 'revenue': revenue, 'profit': profit}

# Run multiple simulations
results = []
for i in range(10000):
    day_result = simulate_business_day()
    results.append(day_result)

# Convert to DataFrame for analysis
df_results = pd.DataFrame(results)
```

Analyzing Simulation Results

```python
# Basic statistics
mean_profit = df_results['profit'].mean()
std_profit = df_results['profit'].std()

# Risk analysis
```

```
loss_probability = (df_results['profit'] < 0).mean()
profit_range = (df_results['profit'] >= 100) & (df_results['profit'] <=
200)
range_probability = profit_range.mean()

# Percentiles for Value at Risk
var_5 = np.percentile(df_results['profit'], 5)  # Worst 5% scenario
var_95 = np.percentile(df_results['profit'], 95)  # Best 5% scenario
```

Time Series Analysis

Working with Dates

```
# Convert strings to datetime
dates = pd.to_datetime(['2024-01-15', '2024-02-20'])

# Extract date components using .dt accessor
df['month'] = df['date'].dt.month
df['day_of_week'] = df['date'].dt.day_of_week  # 0=Monday, 6=Sunday
df['quarter'] = df['date'].dt.quarter
df['is_month_end'] = df['date'].dt.is_month_end

# Access specific elements
third_month = df['date'].dt.month.iloc[2]  # Third row's month
```

Moving Averages

Definition: Smooth time series by averaging over a window of periods.

```
# Simple moving average
df['ma_7'] = df['sales'].rolling(window=7).mean()  # 7-day average
df['ma_30'] = df['sales'].rolling(window=30).mean()  # 30-day average

# Note: First few values will be NaN due to insufficient data
# Use .dropna() to remove NaN values if needed
clean_data = df.dropna()
```

Basic Forecasting Methods

Naive Forecast

```
def naive_forecast(data, periods=1):
    """Tomorrow = today (simplest baseline)"""
    return [data.iloc[-1]] * periods
```

Moving Average Forecast

```
def moving_average_forecast(data, window=7, periods=1):
    """Forecast using average of last 'window' periods"""
    ma = data.iloc[-window:].mean()
    return [ma] * periods
```

Exponential Smoothing

```
def exponential_smoothing_forecast(data, alpha=0.3, periods=1):
    """Weight recent observations more heavily"""
    forecasts = [data.iloc[0]]  # Start with first value

    # Calculate smoothed values
    for i in range(1, len(data)):
        forecast = alpha * data.iloc[i] + (1 - alpha) * forecasts[-1]
        forecasts.append(forecast)

    # Use last smoothed value for future periods
    return [forecasts[-1]] * periods
```

Alpha parameter:

- $\alpha = 0.9$: Very responsive (trust recent data)
- $\alpha = 0.3$: Balanced (typical default)
- $\alpha = 0.1$: Very stable (smooth out noise)

Forecast Accuracy Metrics

```
def calculate_mae(actual, forecast):
    """Mean Absolute Error - average error size"""
    return np.mean(np.abs(actual - forecast))

def calculate_rmse(actual, forecast):
    """Root Mean Squared Error - penalizes large errors"""
    return np.sqrt(np.mean((actual - forecast) ** 2))
```

When to use:

- MAE: Easier to interpret, same units as data
- RMSE: More sensitive to large errors/outliers

Scheduling

Key Performance Metrics

```
def calculate_metrics(schedule_df):
    """Calculate scheduling performance metrics"""
    return {
        'makespan': schedule_df['completion'].max(),  # Total time
        'avg_flow_time': schedule_df['completion'].mean(),  # Average
completion
        'total_tardiness': np.maximum(0, schedule_df['completion'] -
schedule_df['due']).sum(),
        'late_orders': (schedule_df['completion'] >
schedule_df['due']).sum()
    }
```

Key Concepts

- Slack: Scheduling flexibility = Due Time - Processing Time
- Static Scheduling: Sort all orders first, then process sequentially
- Dynamic Scheduling: Make decisions as orders arrive

Common Scheduling Rules

FIFO (First In, First Out)

Process orders in original sequence (by ID or arrival time).

SPT (Shortest Processing Time)

Process shortest jobs first - minimizes average flow time.

EDD (Earliest Due Date)

Process orders with earliest due dates first - minimizes maximum lateness.

Dynamic vs Static Scheduling

Static: All orders available at time 0, sort once and process. Dynamic: Orders arrive over time, make decisions when machine becomes free.

```
# Dynamic scheduling pattern
def schedule_dynamic(orders):
    scheduled = []
    remaining = [o.copy() for o in orders]
    current_time = 0

    while remaining:
        # Find available orders (arrived by current_time)
        available = [o for o in remaining if o['arrival'] <= current_time]

        # If nothing available, jump to next arrival
        if not available:
            current_time = min(o['arrival'] for o in remaining)
            available = [o for o in remaining if o['arrival'] <=
current_time]

        # Apply scheduling rule (e.g., SPT)
        next_order = min(available, key=lambda x: x['processing'])

        # Schedule and update
        next_order['start'] = current_time
        next_order['completion'] = current_time + next_order['processing']
        current_time = next_order['completion']

        scheduled.append(next_order)
        remaining.remove(next_order)
    return scheduled
```

Routing and Local Search

Distance Calculations

```
# Euclidean distance between two points
def calculate_distance(point1, point2):
    """Calculate distance between (x, y) coordinates"""
    x1, y1 = point1
    x2, y2 = point2
    return np.sqrt((x2 - x1)**2 + (y2 - y1)**2)

# Example
depot = (0, 0)
customer = (3, 4)
dist = calculate_distance(depot, customer)  # Returns 5.0
```

Distance Matrix

Definition: Precompute all pairwise distances for efficiency.

```
def create_distance_matrix(locations):
    """Create matrix where distances[i][j] = distance from i to j"""
    n = len(locations)
    distances = np.zeros((n, n))

    for i in range(n):
        for j in range(n):
            if i != j:
                distances[i][j] = calculate_distance(locations[i], 
locations[j])

    return distances
```

Route Distance Calculation

Critical: Always include return to depot (location 0)!

```
def calculate_route_distance(route, distance_matrix):
    """Calculate total distance for a complete route"""
    total = 0

    # Depot to first location
    total += distance_matrix[0, route[0]]

    # Between consecutive locations
    for i in range(len(route) - 1):
        total += distance_matrix[route[i], route[i+1]]
```

```python
    # Last location back to depot
    total += distance_matrix[route[-1], 0]

    return total
```

Greedy Construction: Nearest Neighbor

Strategy: Always visit closest unvisited location next.

```python
def nearest_neighbor_route(distance_matrix):
    """Build route by always choosing nearest unvisited location"""
    n = len(distance_matrix)
    unvisited = list(range(1, n))  # Skip depot (index 0)
    route = []
    current = 0  # Start at depot

    while unvisited:
        # Find nearest unvisited location
        nearest = min(unvisited, key=lambda x: distance_matrix[current, x])
        route.append(nearest)
        unvisited.remove(nearest)
        current = nearest

    return route
```

Local Search: 2-Opt Improvement

2-Opt Swap: Reverse a segment of the route to eliminate crossings.

```python
def perform_2opt_swap(route, i, j):
    """Reverse segment between positions i and j"""
    # Keep start, reverse middle, keep end
    return route[:i+1] + route[i+1:j+1][::-1] + route[j+1:]

# Example: [1, 2, 3, 4, 5] with swap(1, 3) becomes [1, 2, 4, 3, 5]
```

2-Opt Algorithm: Keep improving until no better swap exists.

```python
def improve_route_2opt(route, distance_matrix):
    """Improve route using 2-opt local search"""
    improved = True
    current_route = route.copy()

    while improved:
        improved = False
        current_dist = calculate_route_distance(current_route,
distance_matrix)

        # Try all possible swaps
        for i in range(len(current_route) - 1):
            for j in range(i + 2, len(current_route)):
                new_route = perform_2opt_swap(current_route, i, j)
                new_dist = calculate_route_distance(new_route,
distance_matrix)

                if new_dist < current_dist:
                    current_route = new_route
                    current_dist = new_dist
                    improved = True
                    break  # Restart search

            if improved:
                break  # Exit outer loop

    return current_route
```

Key Patterns

List Slicing for Route Reversal:

```
route = [1, 2, 3, 4, 5, 6]

# Reverse segment from index 2 to 4
route[:2] + route[2:5][::-1] + route[5:]  # [1, 2, 5, 4, 3, 6]
```

Using min() with key parameter:

```
# Find location with minimum distance
nearest = min(unvisited, key=lambda x: distance_matrix[current, x])
```

Route Representation:

- Route = list of location indices (not including depot)
- Example: `[3, 1, 4, 2]` means visit locations 3 ⟶ 1 ⟶ 4 ⟶ 2 ⟶ return to depot

Multi-Objective Optimization

Dominance and Pareto Frontier

Definition: Solution A dominates B if A is better/equal in ALL objectives AND strictly better in at least one.

```
def is_dominated(solution_idx, data):
    """Check if a solution is dominated by any other solution"""
    current = data.iloc[solution_idx]

    for idx in range(len(data)):
        if idx == solution_idx:
            continue

        other = data.iloc[idx]
        # Example: maximize profit, minimize cost
        # Other dominates if: profit >= AND cost <= (with at least one
strict)
        if (other['profit'] >= current['profit'] and
            other['cost'] <= current['cost'] and
            (other['profit'] > current['profit'] or
             other['cost'] < current['cost'])):
            return True

    return False
```

Finding Pareto Frontier

Pareto Frontier: Set of all non-dominated solutions (the only rational choices).

```
def find_pareto_frontier(data):
    """Return only non-dominated solutions"""
    n = len(data)
    is_pareto = np.ones(n, dtype=bool)

    for i in range(n):
        if not is_pareto[i]:
            continue

        for j in range(n):
            if i == j:
                continue

            # Check if j dominates i (adjust for your objectives)
            if (data.iloc[j]['profit'] >= data.iloc[i]['profit'] and
                data.iloc[j]['cost'] <= data.iloc[i]['cost'] and
                (data.iloc[j]['profit'] > data.iloc[i]['profit'] or
                 data.iloc[j]['cost'] < data.iloc[i]['cost'])):
                is_pareto[i] = False
                break

    return data[is_pareto]
```

Normalization to [0,1]

Critical: Always normalize before combining objectives with different scales!

```
def normalize_column(series):
    """Normalize pandas Series to [0, 1] range"""
    min_val = series.min()
    max_val = series.max()

    if max_val > min_val:
        return (series - min_val) / (max_val - min_val)
    else:
        return pd.Series([0.5] * len(series))
```

```python
# Apply to DataFrame columns
data['cost_norm'] = normalize_column(data['cost'])
data['profit_norm'] = normalize_column(data['profit'])
```

Weighted Sum Scoring

Combine objectives using weights that sum to 1.0.

```python
def calculate_weighted_score(data, weights):
    """
    Calculate weighted score for multiple normalized objectives.

    weights: dict like {'profit': 0.6, 'speed': 0.4}

    Note: For minimization objectives, use (1 - normalized_value)
    """
    score = 0

    # Maximize profit (higher is better)
    score += weights['profit'] * data['profit_norm']

    # Minimize time (lower is better, so flip it)
    score += weights['speed'] * (1 - data['time_norm'])

    return score

# Find best solution
data['score'] = calculate_weighted_score(data, {'profit': 0.6, 'speed':
0.4})
best_idx = data['score'].idxmax()
```

Hard Constraints

Constraint: Must be satisfied (feasibility). Objective: Something to optimize.

```python
# Filter to feasible solutions only
constraint_threshold = 100
feasible = data[data['emissions'] <= constraint_threshold]

# Then find Pareto frontier among feasible solutions
pareto_feasible = find_pareto_frontier(feasible)
```

Complete MOO Workflow

Three-stage process for multi-objective problems:

```python
# Stage 1: Filter by constraints
feasible = data[data['constraint_column'] <= threshold]

# Stage 2: Find Pareto frontier
```

```
pareto = find_pareto_frontier(feasible)

# Stage 3: Select using weighted scoring
pareto['obj1_norm'] = normalize_column(pareto['objective1'])
pareto['obj2_norm'] = normalize_column(pareto['objective2'])

# Calculate scores with your priorities
pareto['score'] = (0.6 * pareto['obj1_norm'] +
                   0.4 * (1 - pareto['obj2_norm']))  # if obj2 is
minimization

# Choose best
best_solution = pareto.loc[pareto['score'].idxmax()]
```

Key Patterns

Objective Direction:

- Maximize: Use `normalized_value` directly in score
- Minimize: Use `(1 - normalized_value)` to flip

Weight Selection:

- Weights must sum to 1.0 (representing 100% of priorities)
- Higher weight = more importance
- Example: `w_cost=0.7, w_speed=0.3` means cost is 70% of priority

Common Mistakes:

- Forgetting to normalize (different scales dominate)
- Not flipping minimization objectives in score
- Applying weights before normalization

Metaheuristics

Simulated Annealing (SA)

Core Idea: Accept worse solutions probabilistically to escape local optima, with decreasing probability over time.

Acceptance Criterion:

```
def accept_move(current_cost, new_cost, temperature):
    """Decide whether to accept a move in SA"""
    if new_cost < current_cost:
        return True  # Always accept improvements
    else:
        # Accept worse moves with probability exp(-delta/T)
        delta = new_cost - current_cost
        probability = math.exp(-delta / temperature)
        return random.random() < probability
```

Complete SA Algorithm

Key Pattern: Track BOTH current solution (explores) AND best solution (never forget best).

```python
def simulated_annealing(initial_solution, cost_function,
                       initial_temp=1000, cooling_rate=0.95, min_temp=1):
    """
    Simulated Annealing template.

    Args:
        initial_solution: Starting solution
        cost_function: Function that evaluates solution quality
        initial_temp: Starting temperature (higher = more exploration)
        cooling_rate: Temperature multiplier (0.9-0.99, higher = slower)
        min_temp: Stop when temperature reaches this value
    """
    # Initialize BOTH current and best
    current = initial_solution.copy()
    current_cost = cost_function(current)

    best = current.copy()
    best_cost = current_cost

    temperature = initial_temp

    while temperature > min_temp:
        # Try multiple neighbors per temperature
        for _ in range(10):
            # Generate neighbor (problem-specific)
            neighbor = generate_neighbor(current)
            neighbor_cost = cost_function(neighbor)

            # Acceptance criterion
            if accept_move(current_cost, neighbor_cost, temperature):
                current = neighbor
                current_cost = neighbor_cost

                # Track best ever found (critical!)
                if current_cost < best_cost:
                    best = current.copy()
                    best_cost = current_cost

        # Cool down (geometric cooling)
        temperature *= cooling_rate

    return best, best_cost
```

Temperature & Cooling

Temperature Controls Exploration:

- High T (e.g., 1000): Accept worse moves ~90% ⟶ Explore widely
- Medium T (e.g., 100): Accept worse moves ~30% ⟶ Balance

- Low T (e.g., 10): Accept worse moves <5% ⟶ Exploit (greedy-like)

Common Cooling Schedules:

```
# Geometric cooling (most common)
temperature = temperature * 0.95  # Multiply by constant (0.9-0.99)

# Linear cooling
temperature = temperature - 5  # Subtract constant

# Exponential cooling
temperature = initial_temp / (1 + iteration)
```

Parameter Guidelines:

- Initial Temperature: Start high enough to accept moves ~80% initially
  - Rule of thumb: $T_0 \approx$ average cost difference between neighbors
- Cooling Rate:
  - Fast: $\alpha = 0.9$ (quick, risk of poor solution)
  - Balanced: $\alpha = 0.95$ (good default)
  - Slow: $\alpha = 0.99$ (thorough, slow)
- Iterations per Temperature: 10-50 neighbors per temperature step

Genetic Algorithm Components

Population-based: Maintain multiple solutions that evolve together.

Selection (Tournament):

```
def tournament_selection(population, costs, tournament_size=3):
    """Select parent via tournament"""
    tournament = random.sample(list(zip(population, costs)),
tournament_size)
    return min(tournament, key=lambda x: x[1])[0]  # Return best from
tournament
```

Crossover (Order Crossover):

```
def crossover(parent1, parent2, crossover_point):
    """Combine two parents to create offspring"""
    # Take first part from parent1, second part from parent2
    child = parent1[:crossover_point] + parent2[crossover_point:]
    return child

# For permutations (like routes), need special order crossover to avoid
duplicates!
```

Mutation:

```
def mutate(solution, mutation_rate=0.1):
    """Randomly modify solution with some probability"""
```

```python
    if random.random() < mutation_rate:
        # Make a small random change
        return make_small_change(solution)
    return solution
```

Population Management:

```python
# Elitism: Always keep best solutions
elite_size = 2
new_population = sorted_population[:elite_size]  # Keep best 2

# Generate rest through selection + crossover + mutation
while len(new_population) < population_size:
    parent1 = tournament_selection(population, costs)
    parent2 = tournament_selection(population, costs)
    child = crossover(parent1, parent2)
    child = mutate(child)
    new_population.append(child)
```

Key Patterns

Neighbor Generation:

- Swap: Exchange two elements (works for most problems)
- Insert: Move one element to different position
- Reverse: Reverse a segment (good for routes)

Stopping Criteria:

- Temperature threshold: When T < 1 (SA)
- No improvement: After N iterations without improvement
- Time limit: Stop after X seconds/minutes
- Iteration limit: Stop after N total iterations

Multi-start Strategy:

```python
def multi_start_metaheuristic(n_starts=10):
    """Run metaheuristic from multiple starting points"""
    best_overall = None
    best_cost_overall = float('inf')

    for _ in range(n_starts):
        initial = generate_random_solution()
        solution, cost = simulated_annealing(initial)

        if cost < best_cost_overall:
            best_overall = solution
            best_cost_overall = cost

    return best_overall, best_cost_overall
```

Bibliography