```
"""
================================================================================
                    PYTHON COMPREHENSIONS & GENERATORS MASTER GUIDE
================================================================================

CONCEPT SUMMARY:

1. LIST COMPREHENSIONS [item for item in iterable if condition]
   - WHAT: A concise way to create lists.
   - WHEN TO USE: When you need to transform or filter an existing iterable into
     a NEW list. It is usually faster and more readable than a manual for-loop.

2. SET COMPREHENSIONS {item for item in iterable if condition}
   - WHAT: Similar to lists but produces a SET (unordered, unique elements).
   - WHEN TO USE: When you need a collection of UNIQUE items and don't care about
     order. Perfect for deduplication.

3. DICTIONARY COMPREHENSIONS {key: value for item in iterable if condition}
   - WHAT: Creates a dictionary from an iterable.
   - WHEN TO USE: When you need to map keys to values efficiently, or transform
     an existing dictionary.

4. GENERATOR EXPRESSIONS (item for item in iterable if condition)
   - WHAT: Creates a generator object that produces items one-by-one (lazily).
   - WHEN TO USE: For LARGE datasets where memory is a concern. It doesn't store
     the whole result in RAM; it computes the next value only when requested.

================================================================================
"""

import math
import itertools
import collections
import functools

print("--- START OF MASTER GUIDE ---\n")

# ================================================================================
# PART 1: CORE COMPREHENSIONS & GENERATORS
# ================================================================================

# --------------------------------------------------------------------------------
# 1. LIST COMPREHENSIONS (9 Examples)
# --------------------------------------------------------------------------------
print("1. List Comprehensions")

# E1: Basic Squares
squares = [x**2 for x in range(10)]
# E2: Filtering Evens
evens = [x for x in range(20) if x % 2 == 0]
# E3: String Modification
upper_fruits = [f.upper() for f in ["apple", "cherry"]]
# E4: If-Else Logic
labels = ["Even" if x % 2 == 0 else "Odd" for x in range(5)]
# E5: Flattening Matrix
flat = [x for row in [[1,2], [3,4]] for x in row]
# E6: Extracting Initials
initials = [word[0] for word in ["Python", "Is", "Great"]]
# E7: List of Tuples (Coordinates)
pairs = [(x, y) for x in range(2) for y in range(2)]
# E8: Filtering Non-Empty Strings
cleaned = [s.strip() for s in ["  hi ", "", "bye  "] if s.strip()]
# E9: Nested List Creation (Identity Matrix)
identity = [[1 if i == j else 0 for j in range(3)] for i in range(3)]

print(f"List Examples Count: 9 (Sample: {identity})")

# --------------------------------------------------------------------------------
# 2. SET COMPREHENSIONS (9 Examples)
# --------------------------------------------------------------------------------
print("\n2. Set Comprehensions")

# E1: Unique characters
unique_chars = {c for c in "abracadabra"}
# E2: Lengths of unique words
word_lens = {len(w) for w in ["apple", "banana", "apple"]}
# E3: Vowels in a string
vowels = {c for c in "education" if c in "aeiou"}
# E4: Case-insensitive unique words
unique_words = {w.lower() for w in ["Python", "PYTHON", "python"]}
```

```
# E5: Square roots of unique numbers
roots = {int(x**0.5) for x in [4, 9, 16, 16]}
# E6: Numbers divisible by 5 or 7
div57 = {x for x in range(100) if x % 5 == 0 or x % 7 == 0}
# E7: Non-digit characters
non_digits = {c for c in "Room 101" if not c.isdigit()}
# E8: Relative primes to 10 (sample)
primes_set = {x for x in range(10) if x % 2 != 0 and x % 5 != 0}
# E9: Set of tuples (unique points)
unique_pts = {(x, x*2) for x in range(5)}

print(f"Set Examples Count: 9 (Sample: {vowels})")

# ------------------------------------------------------------------------------
# 3. DICTIONARY COMPREHENSIONS (9 Examples)
# ------------------------------------------------------------------------------
print("\n3. Dictionary Comprehensions")

# E1: Number -> Square mapping
sq_map = {x: x**2 for x in range(5)}
# E2: Character counts (Frequency)
freq = {c: "banana".count(c) for c in set("banana")}
# E3: Inverting a dictionary
prices = {"a": 1, "b": 2}
inv_prices = {v: k for k, v in prices.items()}
# E4: Filtering items by value
expensive = {k: v for k, v in {"milk": 2, "eggs": 5}.items() if v > 3}
# E5: Mapping word to its length
w_map = {w: len(w) for w in ["Python", "Comp"]}
# E6: Handling missing data (if-else in dict)
raw_data = {"a": 10, "b": None}
cleaned_data = {k: (v if v else 0) for k, v in raw_data.items()}
# E7: Unicode mapping
unicode_map = {c: ord(c) for c in "ABC"}
# E8: Conditional keys/values
even_sq_dict = {x: x**2 for x in range(10) if x % 2 == 0}
# E9: Merging lists into dict
keys = ["name", "age"]; vals = ["Alice", 25]
merged = {keys[i]: vals[i] for i in range(len(keys))}

print(f"Dict Examples Count: 9 (Sample: {freq})")

# ------------------------------------------------------------------------------
# 4. GENERATOR EXPRESSIONS (9 Examples)
# ------------------------------------------------------------------------------
print("\n4. Generator Expressions")

# E1: Huge sum (1M items) - Memory efficient
sum_1m = sum(x for x in range(1000000))
# E2: Lazy factorial generator
fact_gen = (math.factorial(x) for x in range(10))
# E3: Reading file lines lazily (mock concept)
lines_gen = (line.strip() for line in ["line 1\n", "line 2\n"])
# E4: Any/All validation
has_vowel = any(c in "aeiou" for c in "pythn")
# E5: Transforming data for print
output_gen = (f"Item {i}" for i in range(5))
# E6: Lazy string reversal
rev_gen = (c for c in reversed("Python"))
# E7: Filtering infinite-like ranges
large_evens = (x for x in range(10**10) if x % 2 == 0) # Just defined, not computed!
# E8: Generator as argument to min/max
min_val = min(x**2 for x in range(-5, 5))
# E9: Chained generator (Double logic)
doubled_gen = (x * 2 for x in (n for n in range(5)))

print(f"Generator Examples Count: 9 (Sample Sum: {sum_1m})")


# ==============================================================================
# PART 2: RELATED ADVANCED CONCEPTS
# ==============================================================================
print("\n--- PART 2: RELATED CONCEPTS ---\n")

# 1. Itertools
print("1. Itertools (chain, product, combinations, groupby, islice)")
# chain: combine iterables
combined = list(itertools.chain([1, 2], [3, 4]))
# product: cartesian product
prod = list(itertools.product([1, 2], ["A", "B"]))
```

```
# combinations: subsets
comb = list(itertools.combinations([1, 2, 3], 2))
# islice: slicing an iterator
sliced = list(itertools.islice(range(100), 5, 10))
print(f"Itertools Sample: {prod}")

# 2. Map, Filter, Reduce
print("\n2. Map, Filter, Reduce")
nums = [1, 2, 3, 4]
mapped = list(map(lambda x: x + 10, nums))
filtered = list(filter(lambda x: x % 2 == 0, nums))
reduced = functools.reduce(lambda x, y: x * y, nums) # 1*2*3*4 = 24
print(f"Functional Tools: Map={mapped}, Filter={filtered}, Reduce={reduced}")

# 3. Walrus Operator (:=)
print("\n3. Walrus Operator (Assignment Expressions)")
# Useful for reusing a computed value inside a comprehension
data = ["apple", "banana", "kiwi"]
long_names = [name for name in data if (n := len(name)) > 4]
print(f"Walrus Sample: {long_names} (lengths were stored in 'n')")

# 4. Zip & Enumerate
print("\n4. Zip & Enumerate")
names = ["Alice", "Bob"]
scores = [90, 85]
zipped = list(zip(names, scores))
enumerated = list(enumerate(names))
print(f"Zip: {zipped}, Enumerate: {enumerated}")

# 5. Lambda Functions
print("\n5. Lambda Functions")
adder = lambda x, y: x + y
print(f"Lambda Output (10+20): {adder(10, 20)}")

# 6. Collections Module (Counter, defaultdict, deque, namedtuple)
print("\n6. Collections Module")
Point = collections.namedtuple('Point', ['x', 'y'])
pt = Point(1, 2)
c_freq = collections.Counter("mississippi")
d_dict = collections.defaultdict(int)
queue = collections.deque([1, 2, 3])
print(f"Collections: Point={pt}, Counter={c_freq}")

# 7. Generator Functions (yield)
print("\n7. Generator Functions (yield)")
def infinite_count(start=0):
    while True:
        yield start
        start += 1
count_gen = infinite_count(10)
print(f"Yield Sample: {next(count_gen)}, {next(count_gen)}")

# 8. Any() & All()
print("\n8. Any() & All()")
test_list = [True, False, True]
print(f"Any (at least one): {any(test_list)}")
print(f"All (every one): {all(test_list)}")


# ================================================================================
# GRAND FINALE: COMBINING CONCEPTS
# ================================================================================
print("\n--- GRAND FINALE: THE 'PYTHONIC' DATA PIPELINE ---\n")

# Scenario: Processing raw string readings using multiple tools at once.
raw_readings = [
    "temp:25.5:C", "temp:error:C", "humidity:45:H",
    "temp:30.2:C", "pressure:1012:P", "humidity:error:H"
]

# 1. Generator Expression: Processes data lazily (memory efficient)
# 2. Walrus Operator (:=): Splits string and checks length in one go
# 3. Nested Validation: Ensures we only process valid numerical data
pipeline = (
    {"type": parts[0], "value": float(parts[1])}
    for r in raw_readings
    if len(parts := r.split(":")) == 3
    if parts[1].replace('.', '', 1).isdigit()
)
```

```
# 4. Collections.Counter + List Comprehension: Summarize the results
# We consume the generator here to create a summary report
summary = collections.Counter(item["type"] for item in pipeline)

print(f"Final Data Pipeline Report: {dict(summary)}")

"""
MOTIVATION FOR THIS EXAMPLE:
This 'Data Pipeline' pattern demonstrates the true power of Pythonic code:
1. Efficiency: The generator ensures we don't store intermediate lists.
2. Readability: Complex logic (split -> validate -> cast -> dict) is handled in a single
   declarative block rather than a 10-line nested for-loop.
3. Expressiveness: Using the Walrus operator (:=) inside the generator allows us to
   reuse the 'parts' variable for both the filter and the final output without
   re-calculating the split() operation.
"""
```