Chapter 5: Comprehensions and Generators - Complete Guide

The map, zip, and filter Functions

map()

Purpose: Apply a function to every item in an iterable

Syntax: `map(function, iterable, ...)`

Basic Usage:

```
# Square all numbers
numbers = [1, 2, 3, 4]
squared = map(lambda x: x**2, numbers)
# Returns: map object (lazy evaluation)
list(squared)  # [1, 4, 9, 16]

# Convert strings to integers
strings = ['1', '2', '3']
integers = list(map(int, strings))  # [1, 2, 3]
```

Multiple Iterables:

```
# Add corresponding elements
a = [1, 2, 3]
b = [10, 20, 30]
result = list(map(lambda x, y: x + y, a, b))  # [11, 22, 33]
```

Key Points:

- Returns an iterator (lazy evaluation)
- Stops at shortest iterable when using multiple iterables
- Modern Python often prefers comprehensions over map

zip()

Purpose: Combine multiple iterables element-by-element

Syntax: `zip(iterable1, iterable2, ...)`

Basic Usage:

```
names = ['Alice', 'Bob', 'Charlie']
ages = [25, 30, 35]
combined = list(zip(names, ages))
# [('Alice', 25), ('Bob', 30), ('Charlie', 35)]

# Unzipping
names, ages = zip(*combined)
```

Key Points:

- Returns an iterator of tuples
- Stops at shortest iterable
- Perfect for parallel iteration
- Use `zip(*zipped)` to unzip

filter()

Purpose: Keep only items that pass a test

Syntax: `filter(function, iterable)`

Basic Usage:

```
numbers = [1, 2, 3, 4, 5, 6]
evens = list(filter(lambda x: x % 2 == 0, numbers))
# [2, 4, 6]

# Filter None/False values
data = [0, 1, False, True, '', 'hello', None]
truthy = list(filter(None, data))  # [1, True, 'hello']
```

Comprehensions

List Comprehensions

Syntax: `[expression for item in iterable if condition]`

Basic Examples:

```
# Basic
squares = [x**2 for x in range(10)]

# With condition
evens = [x for x in range(20) if x % 2 == 0]

# With transformation and condition
```

```
words = ['hello', 'world', 'python']
caps = [w.upper() for w in words if len(w) > 4]
```

Nested Comprehensions

Purpose: Handle nested loops and complex iterations

Examples:

```
# Flatten a 2D list
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flat = [num for row in matrix for num in row]
# [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Create matrix
matrix = [[i*j for j in range(3)] for i in range(3)]

# Multiple conditions
[(x, y) for x in range(3) for y in range(3) if x != y]
```

Reading Order: Left to right, as if written as nested loops:

```
# This comprehension:
[x*y for x in range(3) for y in range(4)]

# Equals this loop:
result = []
for x in range(3):
    for y in range(4):
        result.append(x*y)
```

Filtering a Comprehension

Syntax: Add `if condition` at the end

Examples:

```
# Simple filter
positive = [x for x in numbers if x > 0]

# Multiple conditions
[x for x in range(100) if x % 2 == 0 if x % 5 == 0]
# Same as: if x % 2 == 0 and x % 5 == 0

# Conditional expression (different!)
[x if x > 0 else 0 for x in numbers]  # ternary inside expression
```

Dictionary Comprehensions

Syntax: `{key_expr: value_expr for item in iterable if condition}`

Examples:

```
# Basic
squares = {x: x**2 for x in range(5)}
# {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

# Swap keys and values
original = {'a': 1, 'b': 2}
swapped = {v: k for k, v in original.items()}

# With filter
positive = {k: v for k, v in data.items() if v > 0}

# From two lists
keys = ['name', 'age', 'city']
values = ['Alice', 25, 'NYC']
person = {k: v for k, v in zip(keys, values)}
```

Set Comprehensions

Syntax: `{expression for item in iterable if condition}`

Examples:

```
# Basic
unique_squares = {x**2 for x in [1, -1, 2, -2, 3]}
# {1, 4, 9} - duplicates removed automatically

# With filter
vowels = {c.lower() for c in text if c.lower() in 'aeiou'}
```

Generators

Generator Functions

Purpose: Create iterators using functions with `yield`

Basic Syntax:

```
def simple_generator():
    yield 1
    yield 2
    yield 3
```

```
gen = simple_generator()
print(next(gen))  # 1
print(next(gen))  # 2
print(next(gen))  # 3
# next(gen) would raise StopIteration
```

Practical Examples:

```
# Infinite sequence
def count_up(start=0):
    n = start
    while True:
        yield n
        n += 1

# Fibonacci sequence
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# Read large file line by line
def read_large_file(file_path):
    with open(file_path, 'r') as f:
        for line in f:
            yield line.strip()
```

Key Points:

- Function pauses at `yield` and resumes on next call
- State is preserved between calls
- Memory efficient for large datasets
- Lazy evaluation

Going Beyond next()

Usage with Loops and Functions:

```
# Using in for loop (most common)
def countdown(n):
    while n > 0:
        yield n
        n -= 1

for i in countdown(5):
    print(i)  # 5, 4, 3, 2, 1
```

```
# Convert to list (materializes all values)
numbers = list(countdown(5))

# Using with sum, max, etc.
total = sum(countdown(10))

# Using with any/all
def has_even(numbers):
    for n in numbers:
        if n % 2 == 0:
            yield True
        else:
            yield False
```

send() method:

```
def echo():
    value = None
    while True:
        value = yield value

gen = echo()
next(gen)  # Prime the generator
gen.send(10)  # Sends 10 into generator, returns 10
```

The yield from Expression

Purpose: Delegate to another generator

Syntax: `yield from iterable`

Examples:

```
# Without yield from
def chain_generators(gen1, gen2):
    for item in gen1:
        yield item
    for item in gen2:
        yield item

# With yield from (cleaner)
def chain_generators(gen1, gen2):
    yield from gen1
    yield from gen2

# Practical example: flatten nested structure
def flatten(nested):
    for item in nested:
        if isinstance(item, list):
```

```
            yield from flatten(item)  # Recursive
        else:
            yield item

nested = [1, [2, [3, 4], 5], 6]
list(flatten(nested))  # [1, 2, 3, 4, 5, 6]
```

Generator Expressions

Syntax: `(expression for item in iterable if condition)`

Examples:

```
# Basic (note parentheses instead of brackets)
squares = (x**2 for x in range(1000000))  # No memory overhead!

# Using directly in functions
total = sum(x**2 for x in range(100))  # Parentheses optional here

# Chaining generators
numbers = range(100)
evens = (x for x in numbers if x % 2 == 0)
squared_evens = (x**2 for x in evens)
```

List Comprehension vs Generator Expression:

```
# List comprehension - creates entire list in memory
list_comp = [x**2 for x in range(1000000)]  # Uses ~8MB

# Generator expression - creates on demand
gen_exp = (x**2 for x in range(1000000))    # Uses ~128 bytes
```

Some Performance Considerations

Memory Usage

```
# BAD: Creates large list in memory
def process_large_file(filename):
    return [process(line) for line in open(filename)]

# GOOD: Generator - processes one line at a time
def process_large_file(filename):
    return (process(line) for line in open(filename))
```

Speed Comparisons

- **List comprehensions**: Faster than map/filter for simple operations
- **Generators**: Best for large datasets or infinite sequences
- **Built-in functions**: Usually fastest (written in C)

Timing Example:

```
import time

# List comprehension (fast, but uses memory)
start = time.time()
result = [x**2 for x in range(1000000)]
print(f"List comp: {time.time() - start}")

# Generator (slower per item, but memory efficient)
start = time.time()
result = (x**2 for x in range(1000000))
for item in result:
    pass
print(f"Generator: {time.time() - start}")
```

Don't Overdo Comprehensions and Generators

When to Use What

Use comprehensions when:

- The operation is simple
- Readability is maintained
- You need the entire result at once
- Working with small to medium datasets

Use regular loops when:

- Logic is complex
- Multiple statements needed per iteration
- Comprehension would be hard to read

Bad (too complex):

```
# Hard to read!
result = [process(x) if x > 0 else default(x) if x < -10 else skip(x)
          for x in data if validate(x) if not is_empty(x)]
```

Better (use regular loop):

```
result = []
for x in data:
    if not validate(x) or is_empty(x):
        continue
    if x > 0:
        result.append(process(x))
    elif x < -10:
        result.append(default(x))
    else:
        result.append(skip(x))
```

Readability Guidelines

- If comprehension spans multiple lines, consider a loop
- Maximum one or two conditions in filter clause
- Nested comprehensions should be simple
- If you need comments to explain it, use a loop

Name Localization

Scope Rules in Comprehensions:

```
# Variables in comprehensions are LOCAL
x = 'outer'
[x for x in range(3)]  # x is local to comprehension
print(x)  # Still 'outer' (unchanged)

# Compare to regular loop (pre-Python 3)
for x in range(3):
    pass
print(x)  # Now 2 (variable leaked)

# Walrus operator (:=) CAN leak out
[y := x**2 for x in range(3)]
print(y)  # 4 (last value assigned)
```

Nested Scope:

```
# Variables from outer comprehension accessible in inner
[(x, y) for x in range(3) for y in range(x)]
# x is accessible when defining range for y
```

Generation Behavior in Built-ins

Many built-ins accept and return iterators:

```
# These return iterators
map(func, iterable)
filter(func, iterable)
zip(iter1, iter2)
enumerate(iterable)
reversed(sequence)
range(start, stop)

# These consume iterators
sum(iterable)
max(iterable)
min(iterable)
any(iterable)
all(iterable)
list(iterable)
tuple(iterable)
set(iterable)
```

Chaining Example:

```
# All lazy evaluation until list()
result = list(
    map(str.upper,
        filter(lambda x: len(x) > 3,
               ['cat', 'elephant', 'dog', 'tiger'])))
# ['ELEPHANT', 'TIGER']

# Same with comprehension
result = [s.upper() for s in ['cat', 'elephant', 'dog', 'tiger']
          if len(s) > 3]
```

One Last Example

Complete Real-World Example:

```
# Process log file: find error lines, extract timestamps
def parse_logs(filename):
    """Generator that yields error timestamps from log file."""
    with open(filename, 'r') as f:
        for line in f:
            if 'ERROR' in line:
                # Extract timestamp (assume first word)
                timestamp = line.split()[0]
                yield timestamp
```

```
# Usage
error_times = parse_logs('app.log')

# Get first 10 errors
first_ten = list(zip(range(10), error_times))

# Or process one at a time
for timestamp in parse_logs('app.log'):
    if should_alert(timestamp):
        send_alert(timestamp)
        break  # Stop after first alert
```

Why this is good:

- Memory efficient (doesn't load entire file)
- Lazy evaluation (can stop early)
- Clean separation of concerns
- Composable with other tools

Summary

Quick Reference

Feature	Syntax	Returns	Use When
List comprehension	`[x for x in iter]`	List	Need all results immediately
Dict comprehension	`{k:v for x in iter}`	Dict	Building dictionaries
Set comprehension	`{x for x in iter}`	Set	Need unique values
Generator expression	`(x for x in iter)`	Generator	Large data, lazy evaluation
Generator function	`def f(): yield x`	Generator	Complex logic, state preservation
map()	`map(f, iter)`	Iterator	Applying function to all items
filter()	`filter(f, iter)`	Iterator	Selecting items by condition
zip()	`zip(iter1, iter2)`	Iterator	Combining parallel sequences

Key Takeaways

1. **Comprehensions** are more Pythonic than map/filter for simple cases
2. **Generators** save memory for large datasets
3. **Don't sacrifice readability** for clever one-liners
4. **Use generators** for data pipelines and streaming
5. **Prefer built-ins** when available (they're optimized)
6. **yield from** simplifies generator delegation
7. **Generator expressions** are great for functional programming patterns