Short Expressions and Lambda Functions

Comprehensive Study Guide for Python Programming

Author: Manus AI
Date: December 2024
Topic: Advanced Python Expressions and Functional Programming

---

Table of Contents

Lambda Functions

Lambda functions are anonymous functions that can be defined inline. They provide a concise way to create small, one-line functions without using the `def` keyword.

Basic Lambda Syntax

The basic syntax for lambda functions is:

```
lambda arguments: expression
```

Simple Lambda Examples

```
# Basic lambda function
square = lambda x: x ** 2
```

```python
print(square(5))  # Output: 25

# Lambda with multiple arguments
add = lambda x, y: x + y
print(add(3, 4))  # Output: 7

# Lambda with conditional logic
max_of_two = lambda a, b: a if a > b else b
print(max_of_two(10, 15))  # Output: 15

# Lambda for string operations
capitalize_first = lambda s: s[0].upper() + s[1:].lower() if s
else ""
print(capitalize_first("hello"))  # Output: "Hello"
```

Lambda vs Regular Functions

```python
# Regular function
def multiply_by_two(x):
    return x * 2

# Equivalent lambda
multiply_by_two_lambda = lambda x: x * 2

# Both produce the same result
print(multiply_by_two(5))        # 10
print(multiply_by_two_lambda(5)) # 10
```

Lambda Functions with Built-in Functions

Lambda functions are particularly powerful when used with built-in functions like `map()`, `filter()`, and `sorted()`.

Using Lambda with map()

```python
# Apply a function to all elements in a list
numbers = [1, 2, 3, 4, 5]

# Square all numbers
squared = list(map(lambda x: x ** 2, numbers))
print(squared)  # [1, 4, 9, 16, 25]

# Convert temperatures from Celsius to Fahrenheit
celsius_temps = [0, 20, 30, 40]
fahrenheit = list(map(lambda c: (c * 9/5) + 32, celsius_temps))
print(fahrenheit)  # [32.0, 68.0, 86.0, 104.0]

# String transformations
```

```python
words = ["hello", "world", "python"]
capitalized = list(map(lambda word: word.capitalize(), words))
print(capitalized)  # ['Hello', 'World', 'Python']
```

Using Lambda with filter()

```python
 # Filter elements based on a condition
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Filter even numbers
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)  # [2, 4, 6, 8, 10]

# Filter strings by length
words = ["cat", "elephant", "dog", "hippopotamus", "ant"]
long_words = list(filter(lambda word: len(word) > 5, words))
print(long_words)  # ['elephant', 'hippopotamus']

# Filter positive numbers
mixed_numbers = [-5, -2, 0, 3, 7, -1, 9]
positives = list(filter(lambda x: x > 0, mixed_numbers))
print(positives)  # [3, 7, 9]
```

Using Lambda with sorted()

```python
 # Sort with custom key functions
students = [
    {"name": "Alice", "grade": 85},
    {"name": "Bob", "grade": 92},
    {"name": "Charlie", "grade": 78}
]

# Sort by grade
by_grade = sorted(students, key=lambda student:
student["grade"])
print(by_grade)
# [{'name': 'Charlie', 'grade': 78}, {'name': 'Alice', 'grade':
85}, {'name': 'Bob', 'grade': 92}]

# Sort by name length
words = ["python", "java", "c", "javascript", "go"]
by_length = sorted(words, key=lambda word: len(word))
print(by_length)  # ['c', 'go', 'java', 'python', 'javascript']

# Sort tuples by second element
coordinates = [(1, 3), (2, 1), (3, 2), (1, 1)]
by_y_coordinate = sorted(coordinates, key=lambda point:
```

```
point[1])
print(by_y_coordinate)  # [(2, 1), (1, 1), (3, 2), (1, 3)]
```

Advanced Lambda Patterns

Lambda with Multiple Arguments

```
 # Lambda with multiple parameters
calculate_area = lambda length, width: length * width
print(calculate_area(5, 3))  # 15

# Lambda with default arguments (using regular function syntax)
def create_multiplier(factor=2):
    return lambda x: x * factor

double = create_multiplier(2)
triple = create_multiplier(3)
print(double(5))  # 10
print(triple(5))  # 15
```

Lambda in Data Processing

```
 # Processing list of dictionaries
employees = [
    {"name": "Alice", "salary": 50000, "department":
"Engineering"},
    {"name": "Bob", "salary": 60000, "department": "Marketing"},
    {"name": "Charlie", "salary": 55000, "department":
"Engineering"}
]

# Extract names
names = list(map(lambda emp: emp["name"], employees))
print(names)  # ['Alice', 'Bob', 'Charlie']

# Filter by department
engineers = list(filter(lambda emp: emp["department"] ==
"Engineering", employees))
print(len(engineers))  # 2

# Calculate total salary for engineering department
eng_salaries = map(lambda emp: emp["salary"],
                  filter(lambda emp: emp["department"] ==
"Engineering", employees))
total_eng_salary = sum(eng_salaries)
print(total_eng_salary)  # 105000
```

List Comprehensions

List comprehensions provide a concise way to create lists based on existing iterables. They are more readable and often more efficient than equivalent for-loops.

Basic List Comprehension Syntax

```
# Basic syntax: [expression for item in iterable]
squares = [x**2 for x in range(5)]
print(squares)  # [0, 1, 4, 9, 16]

# Equivalent for-loop
squares_loop = []
for x in range(5):
    squares_loop.append(x**2)
print(squares_loop)  # [0, 1, 4, 9, 16]
```

List Comprehensions with Conditions

Filtering with if Clause

```
# Syntax: [expression for item in iterable if condition]
numbers = range(10)

# Even numbers only
evens = [x for x in numbers if x % 2 == 0]
print(evens)  # [0, 2, 4, 6, 8]

# Squares of odd numbers
odd_squares = [x**2 for x in numbers if x % 2 == 1]
print(odd_squares)  # [1, 9, 25, 49, 81]

# Words longer than 4 characters
words = ["cat", "elephant", "dog", "python", "ant"]
long_words = [word for word in words if len(word) > 4]
print(long_words)  # ['elephant', 'python']
```

Conditional Expressions in List Comprehensions

```
# Using ternary operator in expression
numbers = range(-5, 6)
abs_values = [x if x >= 0 else -x for x in numbers]
print(abs_values)  # [5, 4, 3, 2, 1, 0, 1, 2, 3, 4, 5]

# Categorize numbers
```

```
categories = ["positive" if x > 0 else "negative" if x < 0 else
"zero"
              for x in numbers]
print(categories)
# ['negative', 'negative', 'negative', 'negative', 'negative',
'zero',
#  'positive', 'positive', 'positive', 'positive', 'positive']

# Transform strings with conditions
words = ["hello", "WORLD", "Python", "CODE"]
normalized = [word.lower() if word.isupper() else word.upper()
for word in words]
print(normalized)  # ['HELLO', 'world', 'PYTHON', 'code']
```

Nested List Comprehensions

Processing 2D Lists

```
 # Flatten a 2D list
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened = [item for row in matrix for item in row]
print(flattened)  # [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Equivalent nested loops
flattened_loop = []
for row in matrix:
    for item in row:
        flattened_loop.append(item)
print(flattened_loop)  # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Creating 2D Lists

```
 # Create a multiplication table
multiplication_table = [[i * j for j in range(1, 6)] for i in
range(1, 6)]
print(multiplication_table)
# [[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15],
#  [4, 8, 12, 16, 20], [5, 10, 15, 20, 25]]

# Create a chess board pattern
chess_board = [["black" if (i + j) % 2 == 0 else "white"
                for j in range(8)] for i in range(8)]
```

Complex Nested Comprehensions

```
# Process nested data structures
students_by_class = [
    [{"name": "Alice", "grade": 85}, {"name": "Bob", "grade":
92}],
    [{"name": "Charlie", "grade": 78}, {"name": "Diana",
"grade": 96}]
]

# Extract all names
all_names = [student["name"] for class_list in students_by_class
             for student in class_list]
print(all_names)  # ['Alice', 'Bob', 'Charlie', 'Diana']

# Get high achievers (grade >= 90)
high_achievers = [student["name"] for class_list in
students_by_class
                  for student in class_list if student["grade"]
>= 90]
print(high_achievers)  # ['Bob', 'Diana']
```

String Processing with List Comprehensions

```
# Character-level processing
sentence = "Hello World"
vowels = [char for char in sentence if char.lower() in "aeiou"]
print(vowels)  # ['e', 'o', 'o']

# Word processing
text = "The quick brown fox jumps over the lazy dog"
words = text.split()
capitalized_words = [word.capitalize() for word in words]
print(capitalized_words)
# ['The', 'Quick', 'Brown', 'Fox', 'Jumps', 'Over', 'The',
'Lazy', 'Dog']

# Filter and transform
long_words_upper = [word.upper() for word in words if len(word)
> 4]
print(long_words_upper)  # ['QUICK', 'BROWN', 'JUMPS']
```

Dictionary and Set Comprehensions

Similar to list comprehensions, Python provides comprehensions for dictionaries and sets.

Dictionary Comprehensions

Basic Dictionary Comprehension

```
 # Syntax: {key_expression: value_expression for item in
iterable}
squares_dict = {x: x**2 for x in range(5)}
print(squares_dict)  # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

# Create word length dictionary
words = ["apple", "banana", "cherry", "date"]
word_lengths = {word: len(word) for word in words}
print(word_lengths)  # {'apple': 5, 'banana': 6, 'cherry': 6,
'date': 4}
```

Dictionary Comprehensions with Conditions

```
 # Filter dictionary creation
numbers = range(10)
even_squares = {x: x**2 for x in numbers if x % 2 == 0}
print(even_squares)  # {0: 0, 2: 4, 4: 16, 6: 36, 8: 64}

# Transform existing dictionary
original_prices = {"apple": 1.20, "banana": 0.80, "cherry":
3.00}
discounted_prices = {item: price * 0.9 for item, price in
original_prices.items()
                     if price > 1.00}
print(discounted_prices)  # {'apple': 1.08, 'cherry': 2.7}
```

Processing Data with Dictionary Comprehensions

```
 # Group data by criteria
students = [
    {"name": "Alice", "grade": 85, "subject": "Math"},
    {"name": "Bob", "grade": 92, "subject": "Science"},
    {"name": "Charlie", "grade": 78, "subject": "Math"},
    {"name": "Diana", "grade": 96, "subject": "Science"}
]

# Create grade lookup by name
grade_lookup = {student["name"]: student["grade"] for student in
students}
print(grade_lookup)  # {'Alice': 85, 'Bob': 92, 'Charlie': 78,
'Diana': 96}

# Filter by subject
```

```
math_students = {student["name"]: student["grade"] for student
in students
                 if student["subject"] == "Math"}
print(math_students)  # {'Alice': 85, 'Charlie': 78}
```

Set Comprehensions

Basic Set Comprehension

```
# Syntax: {expression for item in iterable}
unique_squares = {x**2 for x in range(-5, 6)}
print(unique_squares)  # {0, 1, 4, 9, 16, 25}

# Extract unique characters
sentence = "hello world"
unique_chars = {char for char in sentence if char != ' '}
print(unique_chars)  # {'e', 'd', 'h', 'l', 'o', 'r', 'w'}
```

Set Comprehensions with Conditions

```
# Filter unique values
numbers = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5]
unique_evens = {x for x in numbers if x % 2 == 0}
print(unique_evens)  # {2, 4}

# Process text data
words = ["apple", "banana", "apple", "cherry", "banana", "date"]
unique_long_words = {word for word in words if len(word) > 4}
print(unique_long_words)  # {'apple', 'banana', 'cherry'}
```

Generator Expressions

Generator expressions are similar to list comprehensions but create generators instead of lists, making them memory-efficient for large datasets.

Basic Generator Expressions

```
# Syntax: (expression for item in iterable)
squares_gen = (x**2 for x in range(5))
print(type(squares_gen))  # <class 'generator'>

# Convert to list to see values
print(list(squares_gen))  # [0, 1, 4, 9, 16]
```

```
# Generator expressions are memory efficient
large_squares = (x**2 for x in range(1000000))
# Doesn't create list in memory
```

Using Generator Expressions

```
 # With built-in functions
numbers = range(10)
sum_of_squares = sum(x**2 for x in numbers)
print(sum_of_squares)  # 285

# Find maximum of processed values
words = ["hello", "world", "python", "programming"]
max_length = max(len(word) for word in words)
print(max_length)  # 11

# Check if any condition is met
grades = [85, 92, 78, 96, 88]
has_perfect_score = any(grade == 100 for grade in grades)
print(has_perfect_score)  # False

# Check if all conditions are met
all_passing = all(grade >= 60 for grade in grades)
print(all_passing)  # True
```

Generator Expressions vs List Comprehensions

```
 # Memory usage comparison
import sys

# List comprehension - creates entire list in memory
list_comp = [x**2 for x in range(1000)]
print(f"List size: {sys.getsizeof(list_comp)} bytes")

# Generator expression - creates generator object
gen_exp = (x**2 for x in range(1000))
print(f"Generator size: {sys.getsizeof(gen_exp)} bytes")

# Generator is much smaller in memory!
```

Type Annotations

Type annotations provide hints about the expected types of variables, function parameters, and return values, making code more readable and enabling better tooling support.

Basic Type Annotations

Variable Annotations

```python
# Basic type annotations
name: str = "Alice"
age: int = 25
height: float = 5.6
is_student: bool = True

# List annotations
numbers: list[int] = [1, 2, 3, 4, 5]
names: list[str] = ["Alice", "Bob", "Charlie"]

# Dictionary annotations
grades: dict[str, int] = {"Alice": 85, "Bob": 92}
```

Function Annotations

```python
def greet(name: str) -> str:
    """Greet a person by name."""
    return f"Hello, {name}!"

def add_numbers(a: int, b: int) -> int:
    """Add two integers."""
    return a + b

def calculate_average(numbers: list[float]) -> float:
    """Calculate the average of a list of numbers."""
    return sum(numbers) / len(numbers)
```

Advanced Type Annotations

Using typing Module

```python
from typing import List, Dict, Optional, Union, Tuple, Callable

# Optional types (can be None)
def find_user(user_id: int) -> Optional[str]:
    """Find user by ID, return name or None if not found."""
    users = {1: "Alice", 2: "Bob"}
    return users.get(user_id)

# Union types (multiple possible types)
def process_id(user_id: Union[int, str]) -> str:
    """Process user ID as either int or string."""
    return str(user_id)

# Tuple annotations
def get_coordinates() -> Tuple[float, float]:
    """Return x, y coordinates."""
    return (10.5, 20.3)

# Callable annotations
def apply_operation(numbers: List[int], operation:
Callable[[int], int]) -> List[int]:
    """Apply an operation to all numbers."""
    return [operation(num) for num in numbers]
```

Generic Type Annotations

```python
from typing import TypeVar, Generic

T = TypeVar('T')

def get_first_item(items: List[T]) -> Optional[T]:
    """Get the first item from a list."""
    return items[0] if items else None

# Usage maintains type information
numbers = [1, 2, 3]
first_number: Optional[int] = get_first_item(numbers)

words = ["hello", "world"]
first_word: Optional[str] = get_first_item(words)
```

Type Annotations in Classes

```python
class Student:
    """A student with name and grades."""

    def __init__(self, name: str, student_id: int) -> None:
        self.name: str = name
        self.student_id: int = student_id
        self.grades: Dict[str, float] = {}

    def add_grade(self, subject: str, grade: float) -> None:
        """Add a grade for a subject."""
        self.grades[subject] = grade

    def get_average(self) -> Optional[float]:
        """Calculate average grade."""
        if not self.grades:
            return None
        return sum(self.grades.values()) / len(self.grades)

    def get_subjects(self) -> List[str]:
        """Get list of subjects."""
        return list(self.grades.keys())
```

Functional Programming with Built-ins

Python provides several built-in functions that work well with lambda functions and comprehensions for functional programming patterns.

map(), filter(), and reduce()

Advanced map() Usage

```python
 # Multiple iterables with map
list1 = [1, 2, 3, 4]
list2 = [10, 20, 30, 40]
sums = list(map(lambda x, y: x + y, list1, list2))
print(sums)  # [11, 22, 33, 44]

# Map with different length iterables (stops at shortest)
list3 = [100, 200]
products = list(map(lambda x, y, z: x * y * z, list1, list2,
list3))
print(products)  # [1000, 8000]

# String processing with map
words = ["hello", "world", "python"]
lengths_and_caps = list(map(lambda word: (len(word),
word.upper()), words))
print(lengths_and_caps)  # [(5, 'HELLO'), (5, 'WORLD'), (6,
'PYTHON')]
```

Advanced filter() Usage

```python
 # Complex filtering conditions
students = [
    {"name": "Alice", "age": 20, "grade": 85},
    {"name": "Bob", "age": 19, "grade": 92},
    {"name": "Charlie", "age": 21, "grade": 78},
    {"name": "Diana", "age": 20, "grade": 96}
]

# Multiple conditions
young_high_achievers = list(filter(
    lambda student: student["age"] <= 20 and student["grade"] >=
90,
    students
))
print(young_high_achievers)  # [{'name': 'Diana', 'age': 20,
'grade': 96}]

# Filter with None removal
mixed_data = [1, None, 3, None, 5, 0, 7]
clean_data = list(filter(None, mixed_data))  # Removes falsy
values
print(clean_data)  # [1, 3, 5, 7]

# Custom filter function
def is_prime(n):
    if n < 2:
        return False
    return all(n % i != 0 for i in range(2, int(n**0.5) + 1))

numbers = range(2, 20)
primes = list(filter(is_prime, numbers))
print(primes)  # [2, 3, 5, 7, 11, 13, 17, 19]
```

reduce() Function

```
from functools import reduce

# Basic reduce usage
numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # 120

# Reduce with initial value
sum_with_initial = reduce(lambda x, y: x + y, numbers, 100)
print(sum_with_initial)  # 115 (100 + 15)

# Find maximum using reduce
maximum = reduce(lambda x, y: x if x > y else y, numbers)
print(maximum)  # 5

# Complex reduce operations
words = ["hello", "world", "python", "programming"]
```

```python
longest_word = reduce(lambda x, y: x if len(x) > len(y) else y,
words)
print(longest_word)  # "programming"
```

zip() and enumerate()

Advanced zip() Usage

```python
 # Zip multiple lists
names = ["Alice", "Bob", "Charlie"]
ages = [25, 30, 35]
cities = ["New York", "London", "Tokyo"]

combined = list(zip(names, ages, cities))
print(combined)
# [('Alice', 25, 'New York'), ('Bob', 30, 'London'), ('Charlie',
35, 'Tokyo')]

# Unzip using zip with unpacking
unzipped_names, unzipped_ages, unzipped_cities = zip(*combined)
print(unzipped_names)  # ('Alice', 'Bob', 'Charlie')

# Zip with different lengths
list1 = [1, 2, 3, 4, 5]
list2 = ['a', 'b', 'c']
zipped = list(zip(list1, list2))
print(zipped)  # [(1, 'a'), (2, 'b'), (3, 'c')]

# Create dictionary from two lists
keys = ["name", "age", "city"]
values = ["Alice", 25, "New York"]
person_dict = dict(zip(keys, values))
print(person_dict)  # {'name': 'Alice', 'age': 25, 'city': 'New
York'}
```

enumerate() with Comprehensions

```python
 # Basic enumerate usage
words = ["apple", "banana", "cherry"]
indexed_words = [(i, word) for i, word in enumerate(words)]
print(indexed_words)  # [(0, 'apple'), (1, 'banana'), (2,
'cherry')]

# Enumerate with start parameter
indexed_from_one = [(i, word) for i, word in enumerate(words,
start=1)]
print(indexed_from_one)  # [(1, 'apple'), (2, 'banana'), (3,
'cherry')]
```

```
# Filter with enumerate
long_words_with_index = [(i, word) for i, word in
enumerate(words)
                         if len(word) > 5]
print(long_words_with_index)  # [(1, 'banana'), (2, 'cherry')]
```

Conditional Expressions (Ternary Operator)

The ternary operator provides a concise way to write simple if-else statements in a single line.

Basic Ternary Operator

```
# Syntax: value_if_true if condition else value_if_false
age = 18
status = "adult" if age >= 18 else "minor"
print(status)  # "adult"

# Equivalent if-else statement
if age >= 18:
    status = "adult"
else:
    status = "minor"
```

Ternary Operator in Different Contexts

With Functions

```
def get_absolute_value(x):
    return x if x >= 0 else -x

print(get_absolute_value(-5))  # 5
print(get_absolute_value(3))   # 3

# Multiple ternary operators
def categorize_number(x):
    return "positive" if x > 0 else "negative" if x < 0 else
"zero"

print(categorize_number(5))   # "positive"
print(categorize_number(-3))  # "negative"
print(categorize_number(0))   # "zero"
```

In List Comprehensions

```
numbers = [-3, -1, 0, 2, 5]

# Apply absolute value using ternary
abs_values = [x if x >= 0 else -x for x in numbers]
print(abs_values)  # [3, 1, 0, 2, 5]

# Categorize numbers
categories = ["pos" if x > 0 else "neg" if x < 0 else "zero" for
x in numbers]
print(categories)  # ['neg', 'neg', 'zero', 'pos', 'pos']

# Transform strings conditionally
words = ["hello", "WORLD", "Python", "CODE"]
normalized = [word.lower() if word.isupper() else word.upper()
for word in words]
print(normalized)  # ['HELLO', 'world', 'PYTHON', 'code']
```

With Dictionary Operations

```
# Conditional dictionary values
students = ["Alice", "Bob", "Charlie"]
grades = [85, 92, 78]

grade_status = {name: "Pass" if grade >= 80 else "Fail"
                for name, grade in zip(students, grades)}
print(grade_status)  # {'Alice': 'Pass', 'Bob': 'Pass',
'Charlie': 'Fail'}

# Conditional key-value pairs
data = {"a": 10, "b": -5, "c": 0, "d": 15}
positive_data = {k: v if v > 0 else 0 for k, v in data.items()}
print(positive_data)  # {'a': 10, 'b': 0, 'c': 0, 'd': 15}
```

Advanced Expression Patterns

Chaining Operations

```
# Method chaining with string operations
text = "  Hello World  "
processed = text.strip().lower().replace(" ", "_")
print(processed)  # "hello_world"

# Chaining with list operations
```

```python
numbers = [1, 2, 3, 4, 5]
result = sum(x**2 for x in numbers if x % 2 == 0)
print(result)  # 20 (2^2 + 4^2 = 4 + 16)

# Complex chaining
words = ["apple", "banana", "cherry", "date"]
result = "_".join(word.upper() for word in words if len(word) >
4)
print(result)  # "APPLE_BANANA_CHERRY"
```

Nested Expressions

```python
 # Nested comprehensions for complex data processing
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Transpose matrix
transposed = [[row[i] for row in matrix] for i in
range(len(matrix[0]))]
print(transposed)  # [[1, 4, 7], [2, 5, 8], [3, 6, 9]]

# Filter and transform nested data
data = [
    {"name": "Alice", "scores": [85, 90, 88]},
    {"name": "Bob", "scores": [92, 87, 95]},
    {"name": "Charlie", "scores": [78, 82, 80]}
]

# Get names of students with average > 85
high_performers = [student["name"] for student in data
                   if sum(student["scores"]) /
len(student["scores"]) > 85]
print(high_performers)  # ['Alice', 'Bob']
```

Expression Combinations

```python
 # Combining multiple expression types
students = [
    {"name": "Alice", "age": 20, "grades": [85, 90, 88]},
    {"name": "Bob", "age": 19, "grades": [92, 87, 95]},
    {"name": "Charlie", "age": 21, "grades": [78, 82, 80]}
]

# Complex expression combining filter, map, and ternary
result = {
    student["name"]: "Excellent" if (avg :=
sum(student["grades"]) / len(student["grades"])) >= 90
                    else "Good" if avg >= 80
                    else "Needs Improvement"
    for student in students
    if student["age"] <= 20
}
print(result)  # {'Alice': 'Good', 'Bob': 'Excellent'}
```

Performance Considerations

Memory Efficiency

```
 # Generator expressions vs list comprehensions
import sys

# Memory-efficient generator
gen_squares = (x**2 for x in range(1000000))
print(f"Generator size: {sys.getsizeof(gen_squares)} bytes")

# Memory-intensive list
list_squares = [x**2 for x in range(1000000)]
print(f"List size: {sys.getsizeof(list_squares)} bytes")

# Use generators for large datasets
def process_large_file(filename):
    with open(filename) as f:
        return (line.strip().upper() for line in f if
line.strip())
```

Performance Comparisons

```
 import timeit

# List comprehension vs for loop
def list_comp_test():
    return [x**2 for x in range(1000)]

def for_loop_test():
    result = []
    for x in range(1000):
        result.append(x**2)
    return result

# List comprehension is typically faster
list_comp_time = timeit.timeit(list_comp_test, number=10000)
for_loop_time = timeit.timeit(for_loop_test, number=10000)
```

```
print(f"List comprehension: {list_comp_time:.4f} seconds")
print(f"For loop: {for_loop_time:.4f} seconds")
```

When to Use Each Approach

```
# Use list comprehensions for simple transformations
squares = [x**2 for x in range(10)]

# Use regular loops for complex logic
result = []
for x in range(10):
    if x % 2 == 0:
        # Complex processing
        processed = x**2 + x + 1
        if processed > 10:
            result.append(processed)

# Use generator expressions for large datasets
large_data_gen = (process(item) for item in huge_dataset)

# Use map/filter for functional style with existing functions
processed = list(map(str.upper, filter(str.isalpha, words)))
```

Common Exam Patterns

Pattern 1: Convert For-Loop to Comprehension

```
# Original for-loop
result = []
for x in range(10):
    if x % 2 == 0:
        result.append(x**2)

# Converted to list comprehension
result = [x**2 for x in range(10) if x % 2 == 0]
```

Pattern 2: Lambda with Built-in Functions

```
# Sort by custom criteria
students = [("Alice", 85), ("Bob", 92), ("Charlie", 78)]
sorted_by_grade = sorted(students, key=lambda student:
student[1])

# Filter with lambda
```

```
high_grades = list(filter(lambda student: student[1] >= 85,
students))

# Transform with lambda
names = list(map(lambda student: student[0], students))
```

Pattern 3: Dictionary/Set Comprehensions

```
 # Create lookup dictionary
data = [("apple", 5), ("banana", 6), ("cherry", 6)]
length_dict = {fruit: length for fruit, length in data}

# Filter and transform
expensive_items = {item: price * 1.1 for item, price in
prices.items()
                   if price > 10}
```

Pattern 4: Type Annotations

```
 def process_data(items: List[Dict[str, Union[int, str]]]) ->
Dict[str, float]:
    """Process list of dictionaries and return summary."""
    return {item["name"]: float(item["value"]) for item in
items}
```

Pattern 5: Complex Expressions

```
 # Nested comprehension with conditions
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
filtered_flattened = [item for row in matrix for item in row if
item % 2 == 0]

# Ternary operator in comprehension
processed = [x if x > 0 else 0 for x in data]

# Generator expression with built-in
total = sum(x**2 for x in range(100) if x % 3 == 0)
```

Quick Reference Guide

Lambda Function Syntax

```
lambda arguments: expression
lambda x: x**2
lambda x, y: x + y
lambda x: "positive" if x > 0 else "negative"
```

Comprehension Syntax

```
# List comprehension
[expression for item in iterable if condition]

# Dictionary comprehension
{key_expr: value_expr for item in iterable if condition}

# Set comprehension
{expression for item in iterable if condition}

# Generator expression
(expression for item in iterable if condition)
```

Type Annotation Syntax

```
variable: type = value
def function(param: type) -> return_type:
List[type], Dict[key_type, value_type]
Optional[type], Union[type1, type2]
```

Built-in Function Patterns

```
map(function, iterable)
filter(function, iterable)
sorted(iterable, key=function)
sum(generator_expression)
any(generator_expression)
all(generator_expression)
```

Conclusion

Short expressions and lambda functions are powerful tools in Python that enable concise, readable, and efficient code. Mastering these concepts allows you to write more Pythonic code and is essential for advanced Python programming. These patterns frequently appear in programming exams and are fundamental to functional programming approaches in Python.

Key takeaways: - **Lambda functions** provide anonymous function capability for simple operations - **List comprehensions** offer concise alternatives to for-loops - **Type annotations** improve code readability and enable better tooling - **Generator expressions** provide memory-efficient alternatives to list comprehensions - **Functional programming patterns** with map, filter, and reduce enable elegant solutions

Practice these patterns regularly to become proficient in writing concise, efficient Python code.