Chapter 5: Lambda Functions

Anonymous Functions in Python

CS 101 - Fall 2025

On For Today

Let's explore Python's most elegant function syntax!

Topics covered in today's discussion:

- **What are Lambda Functions?** - Anonymous functions explained
- **Basic Lambda Syntax** - The foundation you need
- **Lambda vs Regular Functions** - When to use each
- **Lambda with Built-in Functions** - `map()`, `filter()`, `sorted()`
- **Real-World Applications** - Practical uses in data processing
- **Best Practices** - Writing clean, readable lambda expressions

Get Ready for the Lambda Functions!

What Are Lambda Functions?

Definition

Lambda functions are small, anonymous functions that can have any number of arguments but can only have one expression. They're perfect for short, simple operations!

Think of them as: Mathematical functions like `f(x) = x² + 1` - simple, direct, and to the point!

Any Limitations to Lambda Functions?

! Important

Limitations:

- Lambda functions are restricted to a single expression.
- They cannot contain statements like assignments, if-else, or for loops within their definition.
- They are primarily used for simple, short operations. For more complex logic, a named def function is more appropriate.

Lambda Functions: The Basics

Basic Syntax

```
# Lambda syntax: lambda arguments: expression

# Regular function
def square(x):
    return x * x

# Lambda equivalent
square_lambda = lambda x: x * x

# Using both
print(square(5))        # Output: 25
print(square_lambda(5)) # Output: 25
```

Key Point: Lambda functions are expressions, not statements - they return a value immediately!

Lambda Functions: Multiple Arguments

More Examples

```
# Multiple arguments
add = lambda x, y: x + y
multiply = lambda x, y, z: x * y * z

# With default arguments
greet = lambda name="World": f"Hello, {name}!"

print(add(3, 5))           # Output: 8
print(multiply(2, 3, 4))   # Output: 24
print(greet())             # Output: Hello, World!
print(greet("Alice"))      # Output: Hello, Alice!
```

Why this works: Lambda functions can handle multiple parameters just like regular functions!

Quick Challenge #1 (2 minutes)

Your Turn: Basic Lambda Practice

Challenge: Create lambda functions for these operations:

1. A lambda that calculates the area of a circle: `* r²`
2. A lambda that converts Fahrenheit to Celsius: `(f - 32) * 5/9`
3. A lambda that finds the maximum of three numbers

Starter Code:

```
import math

# Your lambda functions here
circle_area = lambda r: # Complete this
fahrenheit_to_celsius = lambda f: # Complete this
max_three = lambda a, b, c: # Complete this

# Test your functions
print(circle_area(5))
print(fahrenheit_to_celsius(68))
print(max_three(10, 20, 15))
```

Challenge #1 Solutions

Solutions

```python
import math

# Solution 1: Circle area
circle_area = lambda r: math.pi * r * r

# Solution 2: Fahrenheit to Celsius
fahrenheit_to_celsius = lambda f: (f - 32) * 5/9

# Solution 3: Maximum of three numbers
max_three = lambda a, b, c: max(a, max(b, c))
# Alternative: max_three = lambda a, b, c: max(a, b, c)

# Test results
print(f"Circle area (r=5): {circle_area(5):.2f}")      # 78.54
print(f"68°F in Celsius: {fahrenheit_to_celsius(68)}")  # 20.0
print(f"Max of 10,20,15: {max_three(10, 20, 15)}")     # 20
```

Lambda vs Regular Functions

When to Use Each

Lambda: For simple, one-line operations that you'll use briefly
Regular Functions: For complex logic, multiple statements, or reusable code

Rule of thumb: If you can't explain what the function does in one sentence, use a regular function!

Lambda vs Regular: Comparison

Side-by-Side Comparison

```
#  Good use of lambda - simple, clear
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))

#  Bad use of lambda - too complex
complex_lambda = lambda x: x**2 if x > 0 else -x**2 if x < 0 else 0

#  Better as regular function
def process_number(x):
    if x > 0:
        return x**2
    elif x < 0:
        return -x**2
    else:
        return 0
```

Remember: Lambda functions should be simple and readable!

Lambda with Built-in Functions

The Power Combination

Lambda functions really shine when used with Python's built-in functions like `map()`, `filter()`, and `sorted()`!

Why this matters: These combinations let you process data efficiently with minimal code

Meet Your New Best Friends!

Essential Built-in Functions

`map(function, iterable)`
Applies a function to every item in a list/iterable
Think: "Transform every item"

`filter(function, iterable)`
Keeps only items where function returns `True`
Think: "Keep only items that pass the test"
`sorted(iterable, key=function)`
Returns a new sorted list using function for comparison
Think: "Arrange items by custom criteria"
`list(iterable)`
Converts any iterable (map/filter results) into a list
Think: "Make it a proper list I can print/use"

Pro Tip: `map()` and `filter()` return special objects - use `list()` to see the actual results!

Lambda with map()

Transform All Elements

```
# Transform all elements in a list
numbers = [1, 2, 3, 4, 5]
names = ["alice", "bob", "charlie"]

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

# Capitalize all names
capitalized = list(map(lambda name: name.title(), names))
print(f"Capitalized: {capitalized}")  # ['Alice', 'Bob', 'Charlie']

# Multiple lists
nums1 = [1, 2, 3]
nums2 = [4, 5, 6]
sums = list(map(lambda x, y: x + y, nums1, nums2))
print(f"Sums: {sums}")  # [5, 7, 9]
```

Lambda with filter()

Keep Only What You Want

```
# Filter elements based on condition
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
words = ["apple", "banana", "cherry", "date", "elderberry"]

# Keep only even numbers
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(f"Evens: {evens}")  # [2, 4, 6, 8, 10]

# Keep only long words
long_words = list(filter(lambda word: len(word) > 5, words))
print(f"Long words: {long_words}")  # ['banana', 'cherry', 'elderberry']

# Keep positive numbers
mixed = [-3, -1, 0, 2, 5, -7, 9]
positives = list(filter(lambda x: x > 0, mixed))
print(f"Positives: {positives}")  # [2, 5, 9]
```

Lambda with sorted()

Custom Sorting Logic

```
# Sort with custom criteria
students = [
    {"name": "Alice", "grade": 85},
    {"name": "Bob", "grade": 92},
    {"name": "Charlie", "grade": 78},
    {"name": "Diana", "grade": 96}
]

words = ["banana", "pie", "Washington", "book"]

# Sort students by grade (descending)
by_grade = sorted(students, key=lambda student: student["grade"], reverse=True)
print("Top student:", by_grade[0]["name"])  # Diana

# Sort words by length
by_length = sorted(words, key=lambda word: len(word))
print(f"By length: {by_length}")  # ['pie', 'book', 'banana', 'Washington']

# Sort words by last letter
by_last_letter = sorted(words, key=lambda word: word[-1])
print(f"By last letter: {by_last_letter}")  # ['banana', 'pie', 'book', 'Washington']
```

Quick Challenge #2 (3 minutes)

Your Turn: Lambda with Built-ins

Challenge: Use lambda functions with `map()`, `filter()`, and `sorted()`:

```python
# Given data
temperatures_f = [32, 68, 86, 104, 212]  # Fahrenheit
prices = [10.99, 23.45, 5.67, 45.00, 12.34]
products = [
    {"name": "laptop", "price": 999.99, "rating": 4.5},
    {"name": "mouse", "price": 25.50, "rating": 4.2},
    {"name": "keyboard", "price": 75.00, "rating": 4.8},
    {"name": "monitor", "price": 299.99, "rating": 4.3}
]

# Your tasks:
# 1. Convert temperatures to Celsius using map()
# 2. Find prices under $20 using filter()
# 3. Sort products by rating (highest first) using sorted()
```

Challenge #2 Solutions

Solutions

```python
# 1. Convert temperatures to Celsius
celsius = list(map(lambda f: (f - 32) * 5/9, temperatures_f))
print(f"Celsius: {[round(temp, 1) for temp in celsius]}")
# [0.0, 20.0, 30.0, 40.0, 100.0]

# 2. Find prices under $20
cheap_prices = list(filter(lambda price: price < 20, prices))
print(f"Under $20: {cheap_prices}")  # [10.99, 5.67, 12.34]

# 3. Sort products by rating (highest first)
by_rating = sorted(products, key=lambda p: p["rating"], reverse=True)
print("Best rated:", by_rating[0]["name"])  # keyboard
for product in by_rating:
    print(f"{product['name']}: {product['rating']}")
```

Real-World Lambda Applications

Practical Uses

Lambda functions are everywhere in real Python code! Let's see some practical applications you'll encounter.

Common scenarios: Data processing, web development, GUI programming, and scientific computing

More Helpful Functions!

Additional Built-in Functions

`sum(iterable)`
Adds up all numbers in a list/iterable
Think: "Give me the total of all these numbers"
`len(iterable)`
Returns the count of items in a collection
Think: "How many items are there?"
`max(iterable, key=function)`
Finds the largest item (optionally using key function)
Think: "Which item is the biggest/best?"
`set(iterable)`
Creates a collection with only unique items
Think: "Remove all duplicates"

Fun Fact: These functions work great with the results from `map()` and `filter()`!

Real-World Example

Note

Sales Data Processing

```
# Sales data from a CSV or database
sales_data = [
    {"product": "Laptop", "price": 999.99, "quantity": 2, "discount": 0.1},
    {"product": "Mouse", "price": 25.50, "quantity": 5, "discount": 0.0},
    {"product": "Keyboard", "price": 75.00, "quantity": 3, "discount": 0.05},
    {"product": "Monitor", "price": 299.99, "quantity": 1, "discount": 0.15}
]

# Calculate total revenue with discounts
total_revenue = sum(map(
    lambda sale: sale["price"] * sale["quantity"] * (1 - sale["discount"]),
    sales_data
))
print(f"Total Revenue: ${total_revenue:.2f}")

# Find high-value sales (over $200 after discount)
high_value = list(filter(
    lambda sale: sale["price"] * sale["quantity"] * (1 - sale["discount"]) > 200,
    sales_data
))
print(f"High-value sales: {len(high_value)}")
```

Output:

```
# Sales data from a CSV or database
sales_data = [
    {"product": "Laptop", "price": 999.99, "quantity": 2, "discount": 0.1},
    {"product": "Mouse", "price": 25.50, "quantity": 5, "discount": 0.0},
    {"product": "Keyboard", "price": 75.00, "quantity": 3, "discount": 0.05},
    {"product": "Monitor", "price": 299.99, "quantity": 1, "discount": 0.15}
]

# Calculate total revenue with discounts
total_revenue = sum(map(
    lambda sale: sale["price"] * sale["quantity"] * (1 - sale["discount"]),
    sales_data
))
print(f"Total Revenue: ${total_revenue:.2f}")

# Find high-value sales (over $200 after discount)
high_value = list(filter(
    lambda sale: sale["price"] * sale["quantity"] * (1 - sale["discount"]) > 200,
    sales_data
))
print(f"High-value sales: {len(high_value)}")
```

```
Total Revenue: $2396.22
High-value sales: 3
```

Real-World Example

> **Note**
>
> Web Development

```
# User registration data
users = [
    {"email": "alice@email.com", "age": 25, "active": True},
    {"email": "bob@email.com", "age": 17, "active": False},
    {"email": "charlie@email.com", "age": 30, "active": True},
    {"email": "diana@email.com", "age": 16, "active": True}
]

# Get active adult users
active_adults = list(filter(
    lambda user: user["active"] and user["age"] >= 18,
    users
))

# Extract just the email addresses
adult_emails = list(map(lambda user: user["email"], active_adults))
print("Adult user emails:", adult_emails)

# Sort users by age
by_age = sorted(users, key=lambda user: user["age"])
print("Youngest user:", by_age[0]["email"])
```

Output:

```python
# User registration data
users = [
    {"email": "alice@email.com", "age": 25, "active": True},
    {"email": "bob@email.com", "age": 17, "active": False},
    {"email": "charlie@email.com", "age": 30, "active": True},
    {"email": "diana@email.com", "age": 16, "active": True}
]

# Get active adult users
active_adults = list(filter(
    lambda user: user["active"] and user["age"] >= 18,
    users
))

# Extract just the email addresses
adult_emails = list(map(lambda user: user["email"], active_adults))
print("Adult user emails:", adult_emails)

# Sort users by age
by_age = sorted(users, key=lambda user: user["age"])
print("Youngest user:", by_age[0]["email"])
```

```
Adult user emails: ['alice@email.com', 'charlie@email.com']
Youngest user: diana@email.com
```

Real-World Example

> **Note**
>
> Analytic Computing

```
import math

# Experimental data points
data_points = [
    {"x": 1, "y": 2.1, "error": 0.1},
    {"x": 2, "y": 4.2, "error": 0.2},
    {"x": 3, "y": 5.8, "error": 0.15},
    {"x": 4, "y": 8.1, "error": 0.25}
]

# Calculate distances from origin
distances = list(map(
    lambda point: math.sqrt(point["x"]**2 + point["y"]**2),
    data_points
))

# Filter points with low error (high precision)
precise_points = list(filter(
    lambda point: point["error"] < 0.2,
    data_points
))

# Sort by significance (y/error ratio)
by_significance = sorted(
    data_points,
    key=lambda point: point["y"] / point["error"],
    reverse=True
)
print("Most significant point:", by_significance[0])
```

Output:

```python
import math

# Experimental data points
data_points = [
    {"x": 1, "y": 2.1, "error": 0.1},
    {"x": 2, "y": 4.2, "error": 0.2},
    {"x": 3, "y": 5.8, "error": 0.15},
    {"x": 4, "y": 8.1, "error": 0.25}
]

# Calculate distances from origin
distances = list(map(
    lambda point: math.sqrt(point["x"]**2 + point["y"]**2),
    data_points
))

# Filter points with low error (high precision)
precise_points = list(filter(
    lambda point: point["error"] < 0.2,
    data_points
))

# Sort by significance (y/error ratio)
by_significance = sorted(
    data_points,
    key=lambda point: point["y"] / point["error"],
    reverse=True
)
print("Most significant point:", by_significance[0])
```

```
Most significant point: {'x': 3, 'y': 5.8, 'error': 0.15}
```

Real-World Example

> **Note**
>
> GUI (Button Click) Event Handling

```python
# Simulating GUI framework (like tkinter)
class Button:
    def __init__(self, text, command=None):
        self.text = text
        self.command = command

    def click(self):
        if self.command:
            self.command()

# Creating buttons with lambda commands
buttons = [
    Button("Save", lambda: print("File saved!")),
    Button("Load", lambda: print("File loaded!")),
    Button("Exit", lambda: print("Goodbye!"))
]

# Dynamic button creation with different actions
for i in range(3):
    button = Button(f"Button {i+1}", lambda num=i: print(f"Clicked button {num+1}"))
    buttons.append(button)

# Simulate button clicks
for button in buttons:
    button.click()
```

Output:

```python
# Simulating GUI framework (like tkinter)
class Button:
    def __init__(self, text, command=None):
        self.text = text
        self.command = command

    def click(self):
        if self.command:
            self.command()

# Creating buttons with lambda commands
buttons = [
    Button("Save", lambda: print("File saved!")),
    Button("Load", lambda: print("File loaded!")),
    Button("Exit", lambda: print("Goodbye!"))
]

# Dynamic button creation with different actions
for i in range(3):
    button = Button(f"Button {i+1}", lambda num=i: print(f"Clicked button {num+1}"))
    buttons.append(button)

# Simulate button clicks
for button in buttons:
    button.click()
```

```
File saved!
File loaded!
Goodbye!
Clicked button 1
Clicked button 2
Clicked button 3
```

Note: We will talk about classes soon!

Challenge #3: Real-World Practice (5 minutes)

Your Turn: E-commerce Data Processing

You're working for an online store. Process this customer data:

```
customers = [
    {"name": "Alice", "age": 28, "purchases": [45.99, 23.50, 67.25], "member": True},
    {"name": "Bob", "age": 35, "purchases": [12.99, 89.00], "member": False},
    {"name": "Charlie", "age": 22, "purchases": [156.00, 45.50, 23.25, 78.90], "member": Tr
    {"name": "Diana", "age": 45, "purchases": [234.50], "member": True}
]

# Your tasks using lambda functions:
# 1. Calculate total spent by each customer
# 2. Find VIP customers (members who spent > $100)
# 3. Sort customers by average purchase amount
# 4. Get names of customers under 30
```

Challenge #3 Solutions

E-commerce Solutions

```
# 1. Calculate total spent by each customer
customers_with_totals = list(map(
    lambda c: {**c, "total_spent": sum(c["purchases"])},
    customers
))

# 2. Find VIP customers (members who spent > $100)
vip_customers = list(filter(
    lambda c: c["member"] and sum(c["purchases"]) > 100,
    customers
))
print("VIP customers:", [c["name"] for c in vip_customers])

# 3. Sort customers by average purchase amount
by_avg_purchase = sorted(
    customers,
    key=lambda c: sum(c["purchases"]) / len(c["purchases"]),
    reverse=True
)
print("Highest avg purchase:", by_avg_purchase[0]["name"])

# 4. Get names of customers under 30
young_customers = list(map(
    lambda c: c["name"],
    filter(lambda c: c["age"] < 30, customers)
))
print("Young customers:", young_customers)
```

Lambda Best Practices

Writing Clean Lambda Functions

Follow these guidelines to write maintainable and readable lambda expressions.

Remember: Code is read more often than it's written - prioritize clarity!

Lambda Best Practices: Do's and Don'ts

Guidelines

DO:

```
# Simple, clear operations
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))
evens = list(filter(lambda x: x % 2 == 0, numbers))

# Short data transformations
users = [{"name": "Alice", "age": 25}]
names = list(map(lambda u: u["name"], users))
```

DON'T:

```
# Too complex for lambda
complex_func = lambda x: x**2 if x > 0 else abs(x) if x < 0 else "zero"

# Multiple statements (impossible in lambda anyway)
# This won't work:
# bad_lambda = lambda x: print(x); return x**2
```

Lambda Limitations and Alternatives

When Lambda Isn't Enough

```
#  Lambda can't do multiple statements
# Need regular function for this:
def process_grade(score):
    print(f"Processing score: {score}")  # Side effect
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    else:
        return "C"

#  Lambda for simple conditions
grade_simple = lambda score: "Pass" if score >= 60 else "Fail"

#  Lambda can't include assignments
# Need regular function:
def calculate_with_logging(x):
    result = x**2 + 2*x + 1  # Assignment
    print(f"Calculated: {result}")
    return result
```

Final Challenge: Lambda Mastery

Ultimate Lambda Challenge (10 minutes)

You're analyzing student performance data. Use lambda functions to solve these tasks:

```
students = [
    {"name": "Alice", "grades": [85, 92, 78, 96], "major": "CS", "year": 3},
    {"name": "Bob", "grades": [76, 84, 88, 92], "major": "Math", "year": 2},
    {"name": "Charlie", "grades": [95, 89, 94, 97], "major": "CS", "year": 4},
    {"name": "Diana", "grades": [82, 79, 86, 91], "major": "Physics", "year": 3},
    {"name": "Eve", "grades": [88, 92, 85, 89], "major": "CS", "year": 1}
]

# Complete these tasks using ONLY lambda functions with map/filter/sorted:
# 1. Add GPA (average grade) to each student record
# 2. Find honor students (GPA >= 90)
# 3. Get CS majors sorted by year (ascending)
# 4. Find the top performer in each year
# 5. Create a summary: {year: [student_names]} for CS majors only
```

Final Challenge: Complete Solution

Master Solutions

```
# 1. Add GPA to each student record
students_with_gpa = list(map(
    lambda s: {**s, "gpa": sum(s["grades"]) / len(s["grades"])},
    students
))

# 2. Find honor students (GPA >= 90)
honor_students = list(filter(
    lambda s: sum(s["grades"]) / len(s["grades"]) >= 90,
    students
))
print("Honor students:", [s["name"] for s in honor_students])

# 3. Get CS majors sorted by year
cs_students_by_year = sorted(
    filter(lambda s: s["major"] == "CS", students),
    key=lambda s: s["year"]
)
print("CS students by year:", [(s["name"], s["year"]) for s in cs_students_by_year])

# 4. Find top performer in each year (bonus: using max with lambda)
years = list(set(map(lambda s: s["year"], students)))
for year in sorted(years):
    year_students = list(filter(lambda s: s["year"] == year, students))
    top_student = max(year_students, key=lambda s: sum(s["grades"]) / len(s["grades"]))
    print(f"Year {year} top: {top_student['name']} (GPA: {sum(top_student['grades'])/len(to
```

Summary: Lambda Functions Mastery

What You've Learned Today

Core Concepts: * Lambda functions are anonymous, single-expression functions * Perfect for simple operations and data transformations * Excellent with `map()`, `filter()`, and `sorted()`

Practical Skills: * Data processing and filtering * Custom sorting logic * Functional programming patterns * Real-world application scenarios

Best Practices: * Keep lambdas simple and readable * Use regular functions for

complex logic * Prioritize code clarity over cleverness

Congratulations! You've mastered Python's lambda functions!