## Functions

Functions are reusable blocks of code that perform specific tasks.

### Defining and Calling Functions

*# Defining a function*  
**def** greet():  
 print("Hello, World!")  
  
*# Calling a function*  
greet()

### Functions with Parameters

*# Function with parameters*  
**def** greet\_person(name):  
 print(f"Hello, {name}!")  
  
greet\_person("Alice")  
  
*# Function with default parameter*  
**def** greet\_with\_time(name, time="morning"):  
 print(f"Good {time}, {name}!")  
  
greet\_with\_time("Bob") *# Uses default time*  
greet\_with\_time("Bob", "evening") *# Overrides default*

### Return Values

*# Function that returns a value*  
**def** add(a, b):  
 **return** a + b  
  
sum\_result = add(5, 3) *# 8*  
  
*# Function that returns multiple values*  
**def** get\_user\_info():  
 **return** "Alice", 30, "New York"  
  
name, age, city = get\_user\_info() *# Unpacking the tuple*

### Scope of Variables

*# Global variable*  
message = "Hello"  
  
**def** display\_message():  
 *# Local variable*  
 local\_var = "World"  
 print(message + " " + local\_var)  
  
display\_message()  
*# print(local\_var) # This would cause an error - local\_var is not defined outside the function*  
  
*# Modifying global variable inside a function*  
**def** change\_message():  
 **global** message  
 message = "Hi"  
  
change\_message()  
print(message) *# "Hi"*

## Introduction to Lambda Functions

Lambda functions (also called anonymous functions) are small, inline functions defined with the lambda keyword rather than the standard def keyword. They are designed for creating simple functions on-the-fly, particularly when you need a function for a short period and don’t want to formally define it with a name.

## Basic Syntax

**lambda** arguments: expression

Where: - lambda is the Python keyword - arguments are input parameters (can be multiple, comma-separated) - expression is a single expression that gets evaluated and returned

The key characteristic of lambda functions is that they can only contain expressions, not statements. This means they are limited to a single line of code.

## Comparing Regular Functions vs. Lambda Functions

### Regular Function:

**def** square(x):  
 **return** x \*\* 2  
  
result = square(5) *# 25*

### Equivalent Lambda Function:

square = **lambda** x: x \*\* 2  
  
result = square(5) *# 25*

### Multiple Parameters:

*# Regular function*  
**def** multiply(x, y):  
 **return** x \* y  
  
*# Lambda equivalent*  
multiply = **lambda** x, y: x \* y  
  
print(multiply(3, 4)) *# 12*

### No Parameters:

*# Regular function*  
**def** say\_hello():  
 **return** "Hello, World!"  
  
*# Lambda equivalent*  
say\_hello = **lambda**: "Hello, World!"  
  
print(say\_hello()) *# "Hello, World!"*

## Lambda Functions with map()

The map() function applies a given function to each item in an iterable (like a list) and returns a map object (which is an iterator).

### Syntax:

map(function, iterable, ...)

### Examples:

#### Converting Celsius to Fahrenheit for a list of temperatures:

celsius = [0, 10, 20, 30, 40]  
  
*# Using regular function*  
**def** to\_fahrenheit(c):  
 **return** (c \* 9/5) + 32  
  
fahrenheit = list(map(to\_fahrenheit, celsius))  
  
*# Using lambda function*  
fahrenheit = list(map(**lambda** c: (c \* 9/5) + 32, celsius))  
  
print(fahrenheit) *# [32.0, 50.0, 68.0, 86.0, 104.0]*

#### Processing multiple iterables:

*# Add corresponding elements from two lists*  
list1 = [1, 2, 3]  
list2 = [10, 20, 30]  
  
result = list(map(**lambda** x, y: x + y, list1, list2))  
print(result) *# [11, 22, 33]*

#### Applying a function to each character in a string:

*# Convert each character to its ASCII value*  
text = "hello"  
ascii\_values = list(map(**lambda** c: ord(c), text))  
print(ascii\_values) *# [104, 101, 108, 108, 111]*  
  
*# This could also be written as:*  
ascii\_values = list(map(ord, text)) *# Using the built-in ord function directly*

## Lambda Functions with filter()

The filter() function constructs an iterator from elements of an iterable for which a function returns True.

### Syntax:

filter(function, iterable)

### Examples:

#### Filtering even numbers:

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
  
*# Using regular function*  
**def** is\_even(num):  
 **return** num % 2 == 0  
  
even\_numbers = list(filter(is\_even, numbers))  
  
*# Using lambda function*  
even\_numbers = list(filter(**lambda** x: x % 2 == 0, numbers))  
  
print(even\_numbers) *# [2, 4, 6, 8, 10]*

#### Filtering out empty strings:

strings = ["hello", "", "world", "", "python"]  
non\_empty = list(filter(**lambda** s: s != "", strings))  
print(non\_empty) *# ["hello", "world", "python"]*  
  
*# This could also be written as:*  
non\_empty = list(filter(bool, strings)) *# Empty strings are falsy*

#### Finding elements greater than a threshold:

scores = [75, 82, 94, 65, 88, 72, 91, 60]  
passed = list(filter(**lambda** score: score >= 70, scores))  
print(passed) *# [75, 82, 94, 88, 72, 91]*

#### Filtering dictionaries based on values:

students = [  
 {"name": "Alice", "score": 85},  
 {"name": "Bob", "score": 68},  
 {"name": "Charlie", "score": 92},  
 {"name": "David", "score": 73}  
]  
  
*# Filter students with scores above 75*  
high\_achievers = list(filter(**lambda** student: student["score"] > 75, students))  
print(high\_achievers)   
*# [{"name": "Alice", "score": 85}, {"name": "Charlie", "score": 92}]*

## Lambda Functions with sorted()

The sorted() function returns a new sorted list from the items in an iterable. It accepts a key parameter that specifies a function to be called on each list element prior to making comparisons.

### Syntax:

sorted(iterable, key=None, reverse=False)

### Examples:

#### Sorting a list of numbers:

numbers = [5, 2, 9, 1, 7]  
  
*# Basic sorting (no lambda needed)*  
sorted\_numbers = sorted(numbers)  
print(sorted\_numbers) *# [1, 2, 5, 7, 9]*  
  
*# Sorting in descending order*  
sorted\_desc = sorted(numbers, reverse=True)  
print(sorted\_desc) *# [9, 7, 5, 2, 1]*  
  
*# Sorting by absolute value*  
numbers = [-5, 2, -9, 1, 7]  
sorted\_abs = sorted(numbers, key=**lambda** x: abs(x))  
print(sorted\_abs) *# [1, 2, -5, 7, -9]*

#### Sorting strings:

fruits = ["apple", "banana", "cherry", "date", "blueberry"]  
  
*# Sort by length*  
sorted\_by\_length = sorted(fruits, key=**lambda** s: len(s))  
print(sorted\_by\_length)   
*# ["date", "apple", "cherry", "banana", "blueberry"]*  
  
*# Sort by last character*  
sorted\_by\_last = sorted(fruits, key=**lambda** s: s[-1])  
print(sorted\_by\_last)  
*# ["banana", "apple", "date", "blueberry", "cherry"]*

#### Sorting dictionaries:

students = [  
 {"name": "Alice", "score": 85, "age": 20},  
 {"name": "Bob", "score": 92, "age": 22},  
 {"name": "Charlie", "score": 78, "age": 19},  
 {"name": "David", "score": 85, "age": 21}  
]  
  
*# Sort by score*  
sorted\_by\_score = sorted(students, key=**lambda** student: student["score"])  
  
*# Sort by score (descending), then by age (ascending) for tie-breaking*  
sorted\_complex = sorted(  
 students,   
 key=**lambda** student: (-student["score"], student["age"])  
)  
  
*# Sort by name*  
sorted\_by\_name = sorted(students, key=**lambda** student: student["name"])

#### Sorting tuples:

data = [("Alice", 25), ("Bob", 20), ("Charlie", 22)]  
  
*# Sort by second element (age)*  
sorted\_by\_age = sorted(data, key=**lambda** x: x[1])  
print(sorted\_by\_age)   
*# [("Bob", 20), ("Charlie", 22), ("Alice", 25)]*

## Lambda Functions with Custom Sorting

### Multi-level sorting:

*# Sort by length, then alphabetically for ties*  
words = ["apple", "banana", "cat", "dog", "elephant", "fox", "goat"]  
sorted\_words = sorted(words, key=**lambda** s: (len(s), s))  
print(sorted\_words)   
*# ["cat", "dog", "fox", "goat", "apple", "banana", "elephant"]*

### Case-insensitive sorting:

names = ["Alice", "bob", "Charlie", "david"]  
sorted\_names = sorted(names, key=**lambda** s: s.lower())  
print(sorted\_names) *# ["Alice", "bob", "Charlie", "david"]*

## Lambda Functions with reduce()

The reduce() function (from functools module) applies a function of two arguments cumulatively to the items of a sequence, reducing it to a single value.

**from** functools **import** reduce  
  
*# Calculating the product of all numbers in a list*  
numbers = [1, 2, 3, 4, 5]  
product = reduce(**lambda** x, y: x \* y, numbers)  
print(product) *# 120 (1\*2\*3\*4\*5)*  
  
*# Finding the maximum value*  
maximum = reduce(**lambda** x, y: x **if** x > y **else** y, numbers)  
print(maximum) *# 5*

## Advanced Examples

### Combining map() and filter():

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
  
*# Square of even numbers*  
result = list(map(**lambda** x: x\*\*2, filter(**lambda** x: x % 2 == 0, numbers)))  
print(result) *# [4, 16, 36, 64, 100]*  
  
*# Alternatively, using list comprehension*  
*# result = [x\*\*2 for x in numbers if x % 2 == 0]*

### Using lambda with key functions:

*# Find the longest string*  
words = ["apple", "banana", "cherry", "date", "elderberry"]  
longest = max(words, key=**lambda** s: len(s))  
print(longest) *# "elderberry"*  
  
*# Find the smallest absolute value*  
numbers = [5, -10, 3, -8, 2]  
smallest\_abs = min(numbers, key=**lambda** x: abs(x))  
print(smallest\_abs) *# 2*

### Grouping data:

**from** itertools **import** groupby  
  
*# Group words by their first letter*  
words = ["apple", "banana", "cherry", "date", "avocado", "blueberry"]  
sorted\_words = sorted(words, key=**lambda** x: x[0]) *# Sort first for groupby to work*  
  
**for** key, group **in** groupby(sorted\_words, key=**lambda** x: x[0]):  
 print(key, ":", list(group))  
*# a : ["apple", "avocado"]*  
*# b : ["banana", "blueberry"]*  
*# c : ["cherry"]*  
*# d : ["date"]*

## Best Practices for Lambda Functions

### When to Use Lambda Functions:

* Simple, one-line functions
* Functions you’ll use only once (e.g., as arguments to higher-order functions)
* When the function’s purpose is obvious from context

### When to Avoid Lambda Functions:

* Complex operations that would be clearer with a named function
* Functions you need to reuse throughout your code
* Functions that require docstrings or type hints
* When you need to use statements or multiple expressions

### Readability Considerations:

* Lambda functions can make code more concise, but sometimes at the cost of readability
* If a lambda becomes complex, consider using a regular function instead
* Use consistent formatting for complex lambda expressions

## Limitations of Lambda Functions

Lambda functions in Python have several limitations:

1. **Single Expression Only**: Lambda functions can only contain a single expression, not multiple statements.
2. **No Assignment Statements**: You cannot use assignment (=) within a lambda function.
3. **No Annotations**: Lambda functions do not support type hints or annotations.
4. **No Docstrings**: You cannot add documentation strings to explain the function.
5. **Limited Error Messages**: Error messages for lambda functions are often less helpful than for named functions.

*# This is valid:*  
square = **lambda** x: x\*\*2  
  
*# This would be invalid - multiple statements:*  
*# square = lambda x: result = x\*\*2; return result*  
  
*# This would be invalid - assignment:*  
*# square = lambda x: y = x\*\*2; y*

## Conclusion

Lambda functions are a powerful feature in Python for creating small, one-line functions on-the-fly. They are particularly useful when working with higher-order functions like map(), filter(), and sorted(). While they can make your code more concise, it’s important to use them judiciously and not at the expense of readability. For complex operations, a regular named function is often the better choice.

Remember the Zen of Python: “Explicit is better than implicit” and “Simple is better than complex.” Use lambda functions where they help make your code clearer, not more obscure.

**Exercise 12**: Create a simple calculator program using functions for each operation (add, subtract, multiply, divide), then create a main function that asks the user for an operation and two numbers.