PYTHON FUNCTIONS

DR. FRANCIS JESMAR P. MONTALBO

OUTLINE

- User-defined Functions
- Built-in Functions
- Lambda Functions
- Higher-order Functions

USER-DEFINED FUNCTIONS

WHAT ARE USER-DEFINED FUNCTIONS?

- A way to organize and reuse code.
- Creates a block of code once and use it multiple times in your program.
- Defined using the "def" keyword, and they can take in parameters and return values.

EXAMPLE

- Here's an example of a simple function that takes in two numbers and returns their sum:

```
def add(num1, num2):
    return num1 + num2

result = add(55, 6)
print(result) # prints 61
```

EXPLANATION

- In the example, the function "add" takes in two parameters, “num1" and “num2", and returns the sum of those two numbers. We then call the function by passing in the values 55 and 6 as the arguments, and the function returns 61 which is the sum of 55 and 6.

EXAMPLE

- Functions can also have default values for their parameters, which means that if a value for that parameter is not provided when the function is called, the default value will be used instead. For example:

```
def add(a, b=0):
    return a + b


result = add(3)
print(result) # prints 3


result = add(3,4)
print(result) # prints 7
```

EXPLANATION

- In the example, the function "add" takes in two parameters, "a" and "b", with a default value of 0 for b. When we call the function with one argument, the default value of b is used.

FUNCTION SCOPE

- Another important concept related to functions is the idea of a "scope". Variables defined inside a function are only accessible within that function and are not available outside of it.

```
def my_function():
    x = 5
    print(x)

my_function() # prints 5
print(x) # raises an error
```

EXPLANATION

- In the example, the variable "x" is defined inside the function "my_function" and is only accessible within that function. Attempting to access it outside of the function will raise an error.

CALLBACK FUNCTION

- Functions can also be used as arguments to other functions. This is called "callback" function, when a function is passed as an argument to another function.

```
def square(x):
    return x*x

def cube(x):
    return x*x*x

def apply(func, x):
    return func(x)

print(apply(square, 3)) # prints 9
print(apply(cube, 3)) # prints 27
```

EXPLANATION

- In the example, we have two functions "square" and "cube", which take a number and return its square or cube respectively. We also have a function "apply" which takes two arguments, first is a function and second is a number, it applies that function on that number.

ARGUMENTS

ARGUMENTS

- Values passed to a function when it is called. These values are then assigned to the function's parameters, which are the names used within the function to refer to the received values.
  - Types of Arguments:
    - Positional Arguments
    - Keyword Arguments
    - Variable-length Arguments

POSITIONAL ARGUMENTS

- In this example, the function "my_function" takes three positional arguments "a", "b", and "c". When we call the function with the values 1, 2, and 3, the function adds them together and returns the result, which is 6.

```
def my_function(a, b, c):
   return a + b + c


result = my_function(1, 2, 3)
print(result) # 6
```

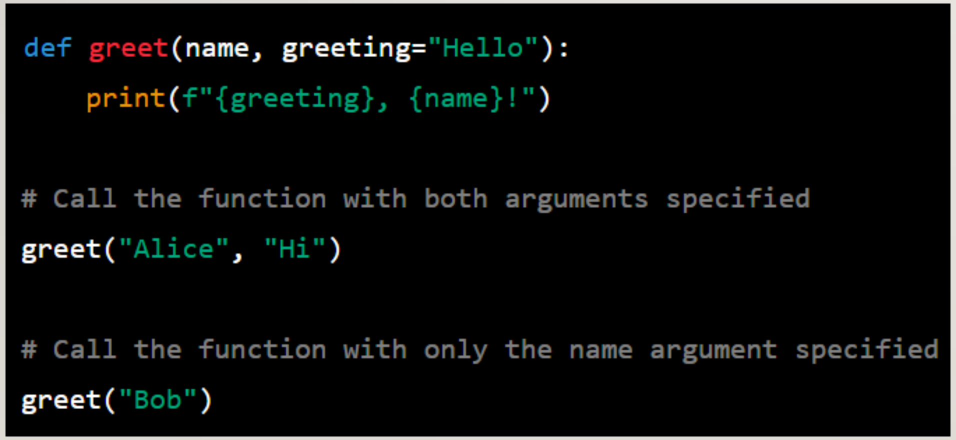
KEYWORD ARGUMENTS

- In this example, the function "my_function" takes three keyword arguments "a", "b", and "c". When we call the function with the keyword arguments c=1, a=2, and b=3, the function adds them together and returns the result, which is 6.

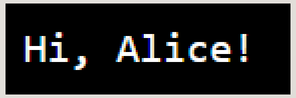
```
def my_function(a, b, c):
    return a + b + c


result = my_function(c=1, a=2, b=3)
print(result) # 6
```

DEFAULT VALUE ARGUMENT

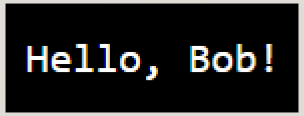
- In Python, you can define a function with default values for its arguments. This means that if the user doesn't supply a value for a particular argument when calling the function, it will automatically be set to the default value specified in the function definition.



```python
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}!")

# Call the function with both arguments specified
greet("Alice", "Hi")

# Call the function with only the name argument specified
greet("Bob")
```



```
Hi, Alice!
```



```
Hello, Bob!
```

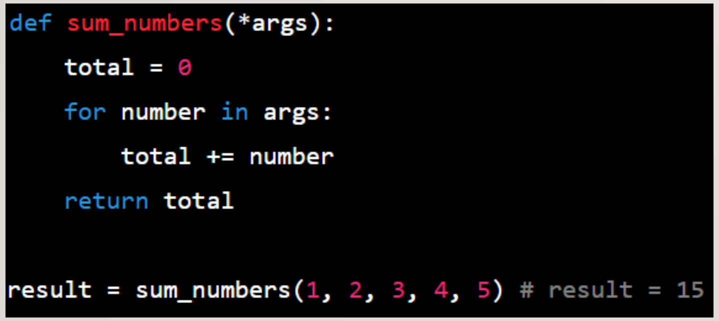
VARIABLE-LENGTH ARGUMENTS

- In this example, the function "my_function" takes a variable-length tuple of arguments "*args". When we call the function with the values 1, 2, 3, 4, and 5, the function adds them together using the built-in "sum" function and returns the result, which is 15.

```
def my_function(*args):
    return sum(args)


result = my_function(1, 2, 3, 4, 5)
print(result) # 15
```

*ARGS

- Use *args when you need to accept an arbitrary number of non-keyword arguments. This is useful when you don't know in advance how many arguments you need to pass to a function, or when you want to make your code more flexible. For example, if you are writing a function that computes the sum of a list of numbers, you could use *args to allow the function to accept any number of arguments:

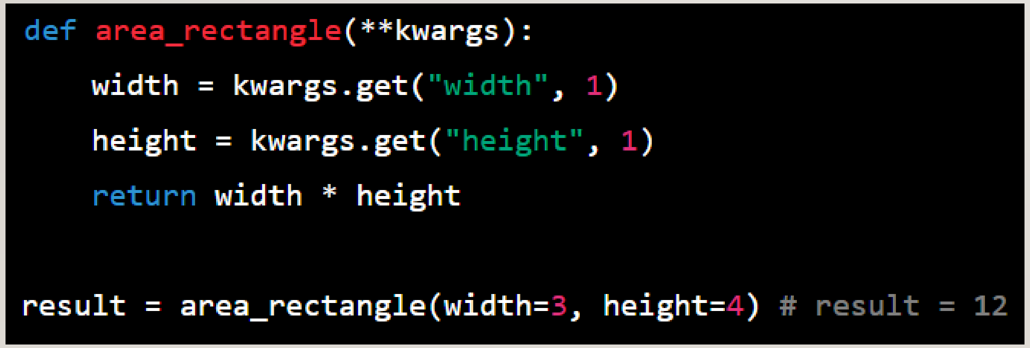


```python
def sum_numbers(*args):
    total = 0
    for number in args:
        total += number
    return total

result = sum_numbers(1, 2, 3, 4, 5) # result = 15
```

When *args is used in a function definition, any non-keyword arguments passed to the function are collected into a **TUPLE**.

**KWARGS

- Use **kwargs when you need to accept an arbitrary number of keyword arguments. This is useful when you want to provide default values for some of the arguments, or when you want to make your code more flexible. For example, if you are writing a function that computes the area of a rectangle, you could use **kwargs to allow the function to accept any combination of width and height, and to provide default values for width and height if they are not provided:



```python
def area_rectangle(**kwargs):
    width = kwargs.get("width", 1)
    height = kwargs.get("height", 1)
    return width * height

result = area_rectangle(width=3, height=4) # result = 12
```

` 1 ` in this example is the default value for 'width' and 'height'

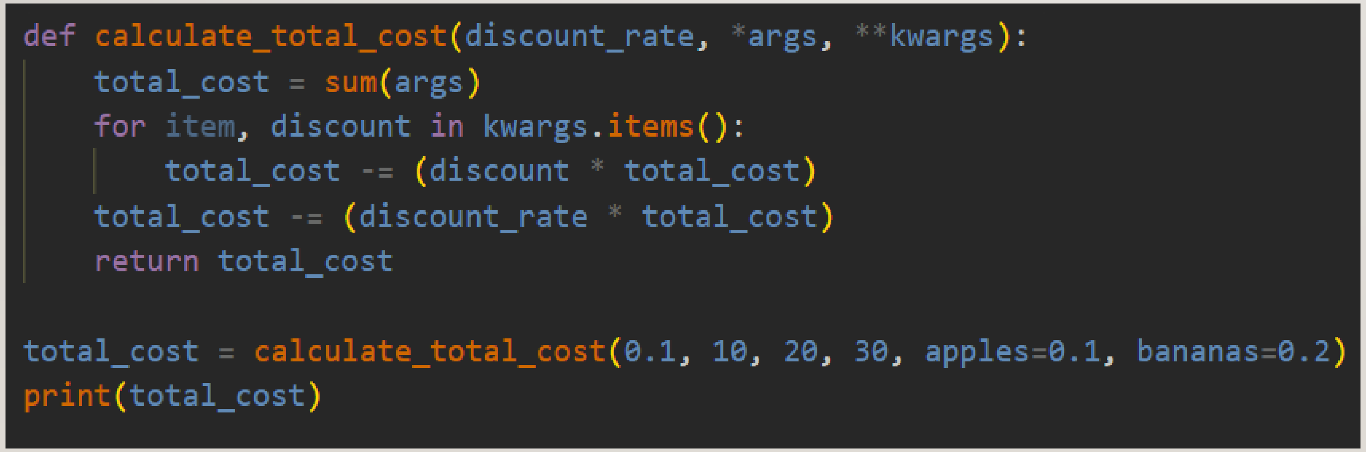
When **kwargs is used in a function definition, any keyword arguments passed to the function are collected into a **DICTIONARY**.

*ARGS AND **KWARGS

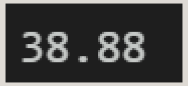
- *args and **kwargs are used as arguments if there is doubt in the number arguments to be passed in a function.
- It is worth noting that the "*" (args) and "**" (kwargs) are the indicators rather than args or kwargs.
- In general, *args and **kwargs are most useful when you don't know in advance how many arguments or keyword arguments you need to pass to a function, or when you want to make your code more flexible.
- They are particularly useful when you are writing functions that work with different types of data, or when you are writing functions that need to be reused in different contexts.

USING BOTH *ARGS AND **KWARGS

- Use *args and **kwargs together when you need to accept both non-keyword and keyword arguments. This is useful when you want to provide maximum flexibility for your function. For example, if you are writing a function that computes the sum of a list of numbers, and you want to allow the user to provide an optional starting value, you could use both *args and **kwargs:



```
def calculate_total_cost(discount_rate, *args, **kwargs):
    total_cost = sum(args)
    for item, discount in kwargs.items():
        total_cost -= (discount * total_cost)
    total_cost -= (discount_rate * total_cost)
    return total_cost

total_cost = calculate_total_cost(0.1, 10, 20, 30, apples=0.1, bananas=0.2)
print(total_cost)
```



```
38.88
```

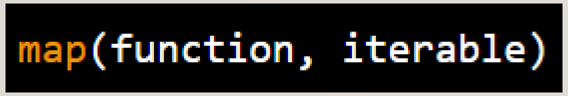
BUILT-IN FUNCTIONS

BUILT-IN FUNCTIONS

- A pre-defined function that is available in the global namespace of the interpreter. These functions are always available to use in any Python program without the need for any additional imports or installations.
- map()
- filter()
- reduce()
- sorted()

MAP()

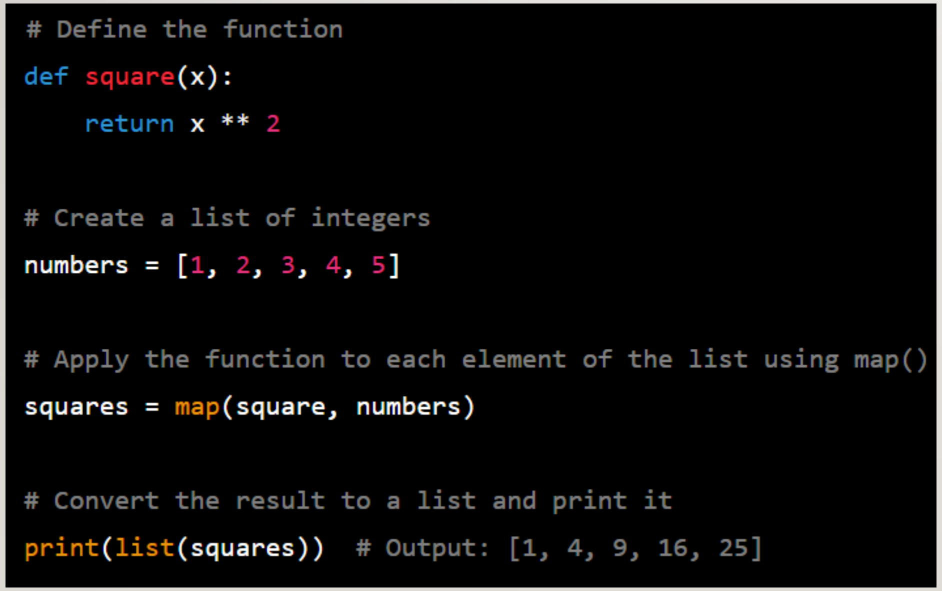
- Executes a specified function for each item in an iterable. The item is sent to the function as a parameter.
- Such elements of an iterable include a list, tuple, or string and returns a new iterable object with the results.
- This built-in function allows you to process and transform all the items in an iterable without using an explicit for loop
- function is a function that takes one argument and returns a value.
- iterable is an iterable object that contains the values to be processed by the function.



```
map(function, iterable)
```

MAP() EXAMPLE

- Here's an example of using map() to convert a list of integers to a list of their squares:



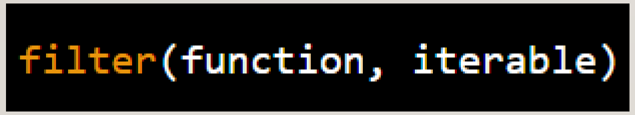
```python
# Define the function
def square(x):
    return x ** 2

# Create a list of integers
numbers = [1, 2, 3, 4, 5]

# Apply the function to each element of the list using map()
squares = map(square, numbers)

# Convert the result to a list and print it
print(list(squares))  # Output: [1, 4, 9, 16, 25]
```

In this example, map() applies the square() function to each element of the numbers list and returns a new iterable containing the squares of the original values.

The list() function is used to convert the resulting iterable to a list so that it can be printed.

FILTER()

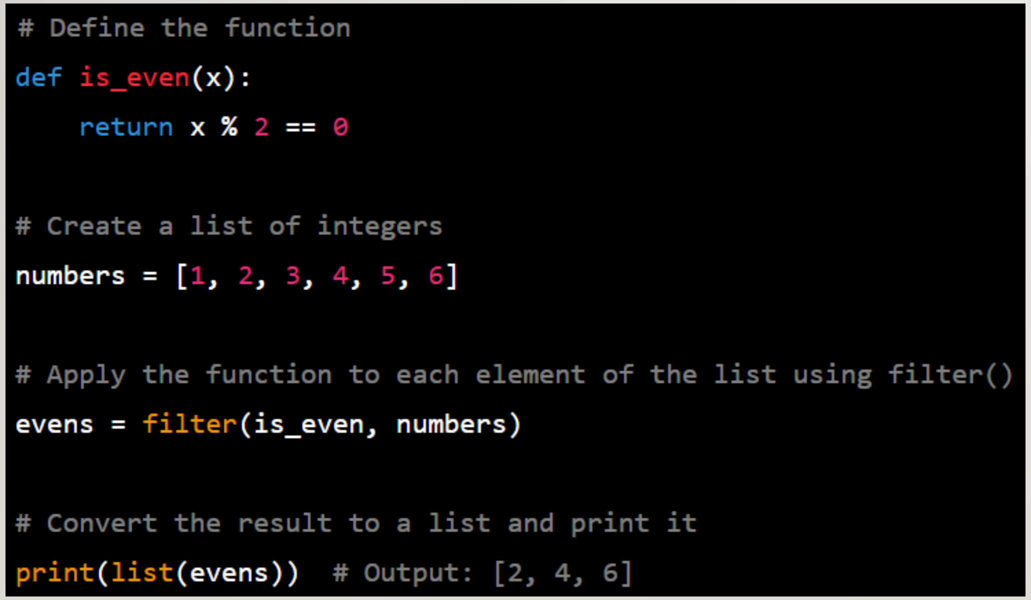
- Filters the elements of an iterable based on a given function and returns an iterator containing only the elements for which the function returns True.
- function is a function that takes one argument and returns a boolean value (True or False).
- This built-in function allows you to process an iterable and extract those items that satisfy a given condition
- iterable is an iterable object that contains the values to be filtered.



```
filter(function, iterable)
```

FILTER() EXAMPLE

- Here's an example of using filter() to keep only the even numbers in a list of integers:



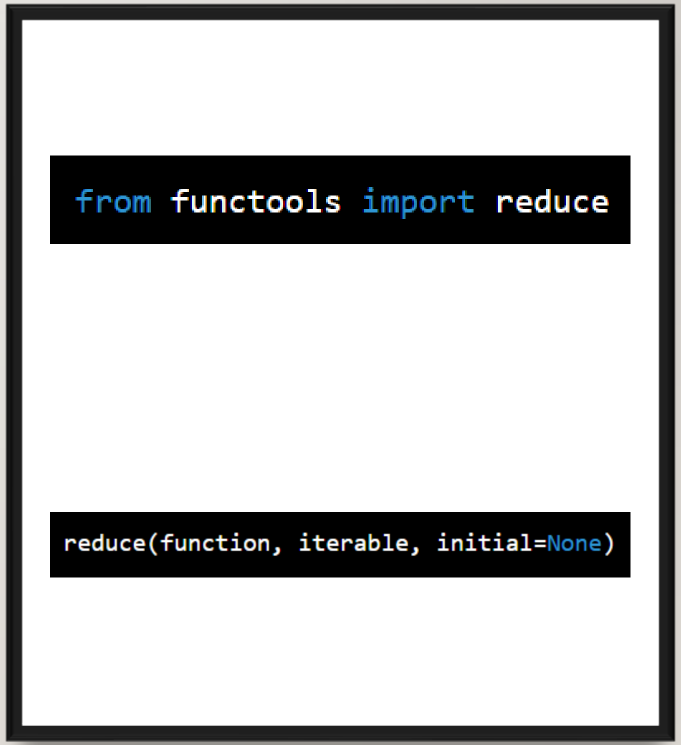
```python
# Define the function
def is_even(x):
    return x % 2 == 0

# Create a list of integers
numbers = [1, 2, 3, 4, 5, 6]

# Apply the function to each element of the list using filter()
evens = filter(is_even, numbers)

# Convert the result to a list and print it
print(list(evens))  # Output: [2, 4, 6]
```

In this example, filter() applies the is_even() function to each element of the numbers list and returns an iterator containing only the even numbers.

The list() function is used to convert the resulting iterator to a list so that it can be printed.

REDUCE()

- Applies a given function to the elements of an iterable in a cumulative way and returns a single value that represents the "reduced" result of the operation.
- function is a function that takes two arguments and returns a value.
- iterable is an iterable object that contains the values to be reduced.
- initial is an optional initial value to start the accumulation. If not provided, the first element of the iterable is used as the initial value.
- In Python 2, reduce() was a built-in function and did not require importing from the functools module. However, in Python 3, reduce() was moved to the functools module and is no longer a built-in function.

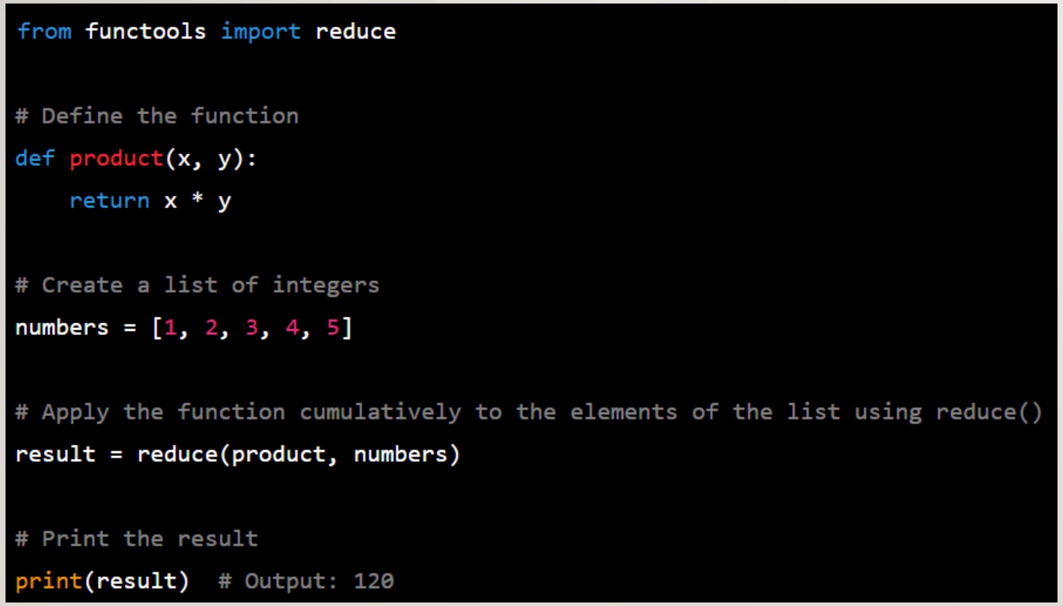


```python
from functools import reduce
```

```python
reduce(function, iterable, initial=None)
```

REDUCE() EXAMPLE

- Here's an example of using reduce() to compute the product of a list of integers:



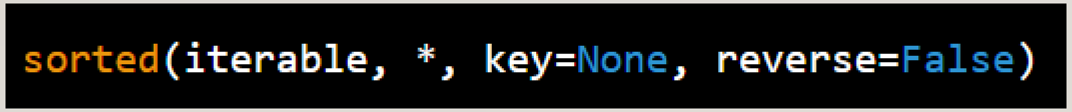
```python
from functools import reduce

# Define the function
def product(x, y):
    return x * y

# Create a list of integers
numbers = [1, 2, 3, 4, 5]

# Apply the function cumulatively to the elements of the list using reduce()
result = reduce(product, numbers)

# Print the result
print(result)  # Output: 120
```

In this example, **reduce()** applies the **product()** function cumulatively to the elements of the numbers list and returns the product of all the elements.

The functools module is used to import reduce() since it is not a built-in function in Python 3.The initial value is not provided, so the first element of the list (1) is used as the initial value

SORTED()

- Takes an iterable as input and returns a new sorted list of the elements in the iterable. The sorted() function can be used with any iterable data type such as lists, tuples, or strings.
- iterable is the iterable to be sorted.
- key is an optional function that is used to determine the sorting order of each element in the iterable. If provided, the function should take an element of the iterable as input and return a value that will be used as the sorting key for that element.
- reverse is an optional boolean flag that specifies whether the elements should be sorted in reverse order.



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

SORTED() EXAMPLE

- Sorting a list of numbers in ascending order:

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
sorted_numbers = sorted(numbers)
print(sorted_numbers)  # Output: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
```

Sorting a list of strings in alphabetical order:

```
words = ['apple', 'banana', 'cherry', 'date', 'elderberry']
sorted_words = sorted(words)
print(sorted_words)  # Output: ['apple', 'banana', 'cherry', 'date', 'elderberry']
```

LAMBDA FUNCTIONS

LAMBDA FUNCTIONS

- A lambda function (also called an "**anonymous function**" or a "**function literal**") is a small, one-line function that is defined without a name.
- Lambda functions are often used when you need to define a simple function that you will only use once, or when you want to pass a function as an argument to another function.
- Lambda functions are defined using the **lambda** keyword, followed by the function's arguments and a colon, and then the expression that should be evaluated and returned by the function.

LAMBDA ADVANTAGES

- Provides a **cleaner** code
- **Performs similarly** to a function
- **Preserves namespaces** and **reduces code pollution**
- **Improve the functionalities of built-in functions** and **make them more expressive** (expressive in programming means concisely readable and meaningful)
- **Fas**t and **requires less overhead** than a user-defined function
- Provides **a dedicated function**

LAMBDA DISADVANTAGES

- Non-re-usable
- Has no exact identity
- Cannot include statements or multiple expressions
- Can make code more difficult to debug if not used properly

ADDITIONAL NOTES TO USING LAMBDA

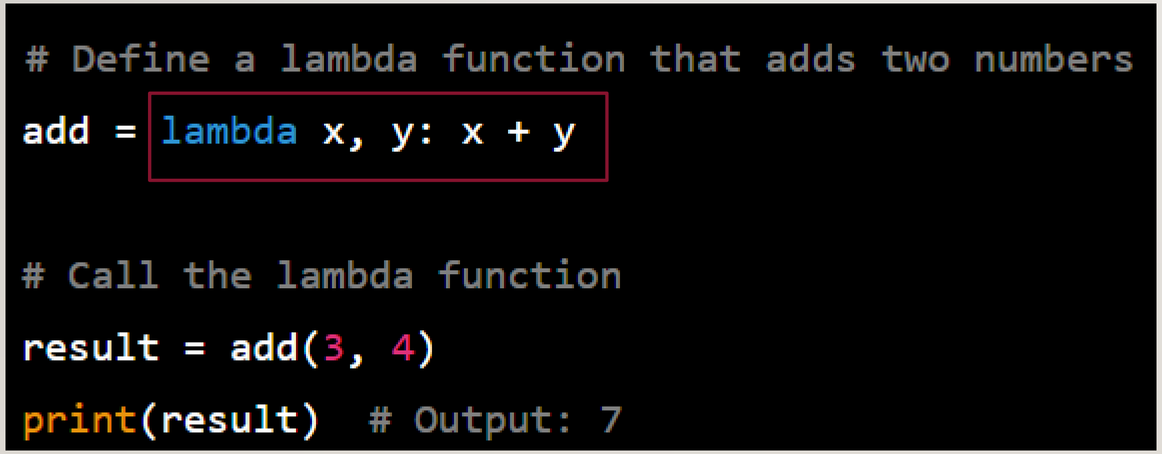
Lambda functions are **limited to a single expression** and **cannot include statements or multiple expressions** because they are designed to be used as small, anonymous functions that can be passed as arguments to other functions.

By restricting lambda functions to a single expression, they can be more concise and easier to use in these contexts.

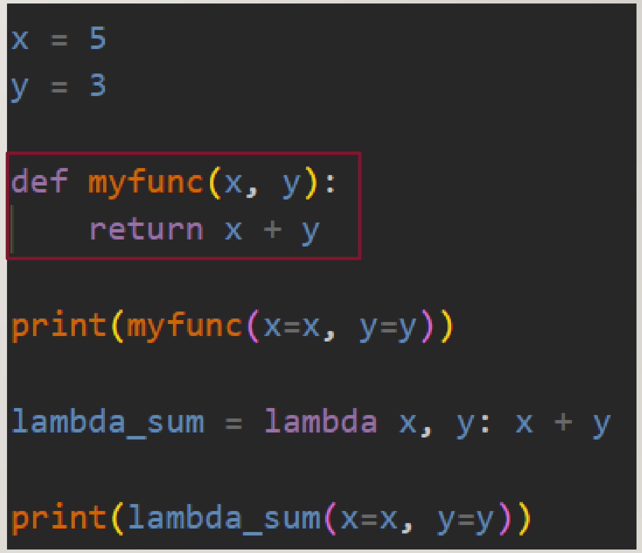
In contrast, user-defined functions and built-in functions can include multiple statements and expressions and can have a more complex structure. This makes them more versatile and better suited to larger or more complex tasks.

LAMBDA EXAMPLES

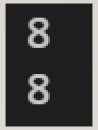
- We define a lambda function called add that takes two arguments x and y and returns their sum. We can then call the lambda function and store the result in a variable result.
- After the ":" indicates the return values for lambda.



```python
# Define a lambda function that adds two numbers
add = lambda x, y: x + y

# Call the lambda function
result = add(3, 4)
print(result)  # Output: 7
```



```python
x = 5
y = 3

def myfunc(x, y):
    return x + y

print(myfunc(x=x, y=y))

lambda_sum = lambda x, y: x + y

print(lambda_sum(x=x, y=y))
```



```
8
8
```

HIGHER-ORDER FUNCTIONS

HIGHER-ORDER FUNCTIONS WITH LAMBDA

A higher-order function is a function that takes one or more functions as arguments, or that returns a function as its result.

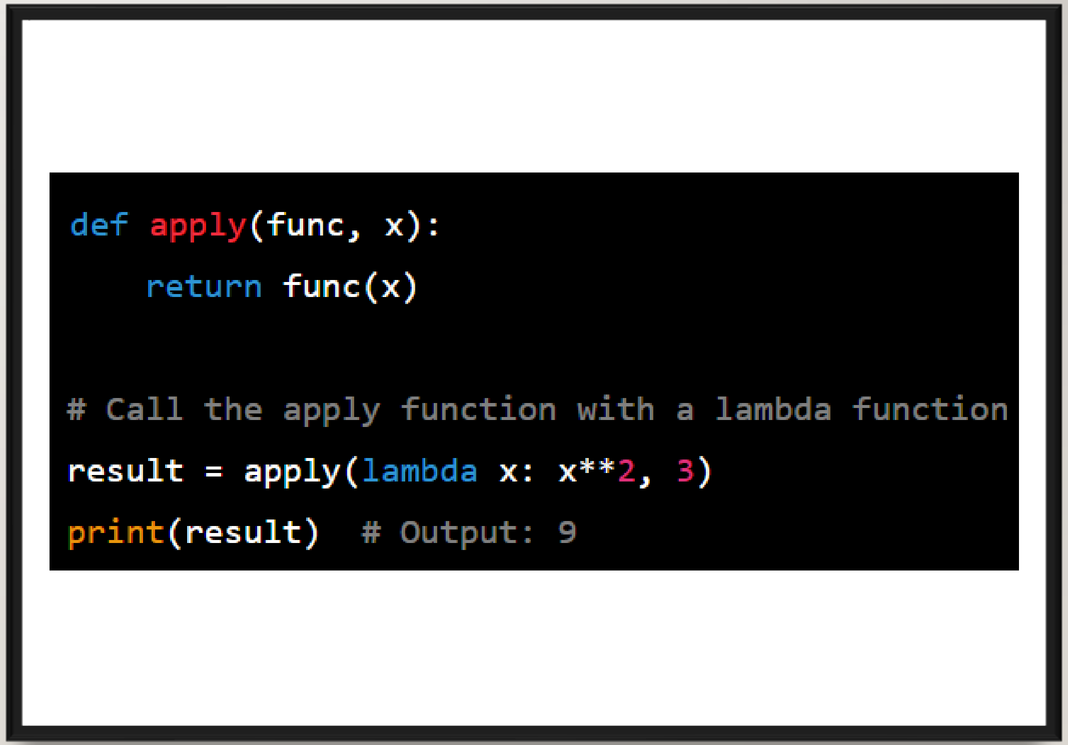
In the context of lambda functions, a higher-order function is a function that takes one or more lambda functions as arguments, or that returns a lambda function as its result.

Higher-order functions and lambda functions can be very powerful tools for writing concise and expressive code in Python.

They allow you to create small, single-purpose functions that can be combined and reused in many different contexts, and they can help you write code that is easier to read, write, and maintain.

HIGHER-ORDER FUNCTIONS WITH LAMBDA

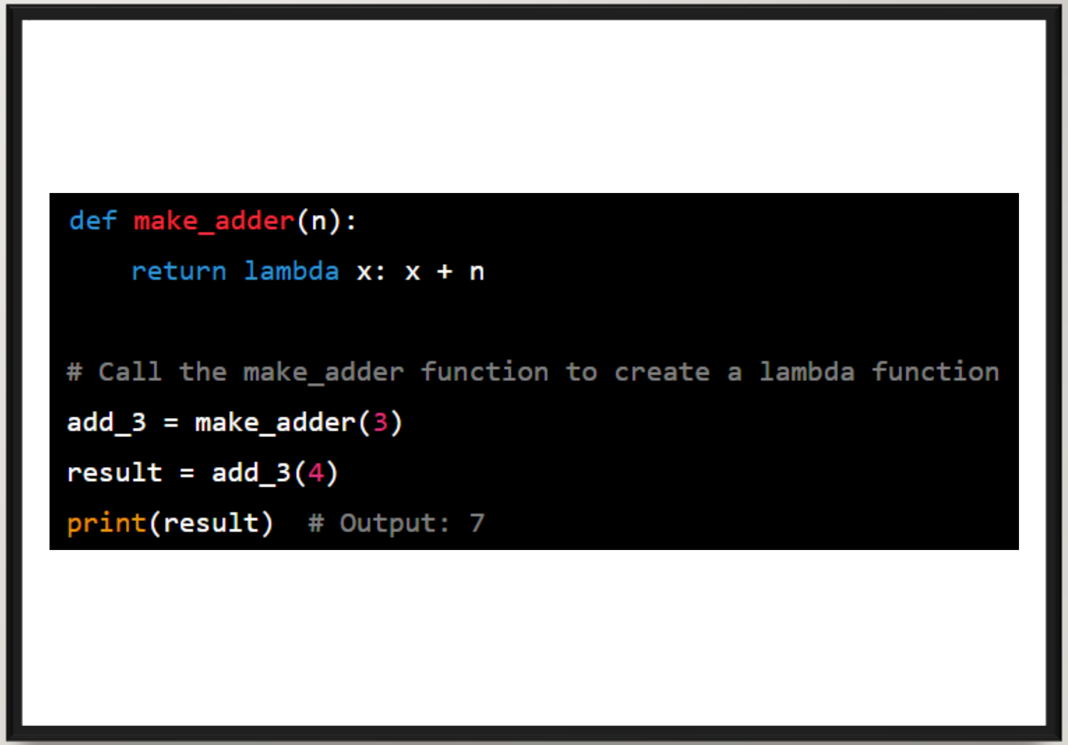
- In this example, we define a higher-order function called apply that takes a function func and an argument x and applies func to x by calling **func(x)** and returning the result. We then call the apply function with a lambda function that squares its argument.



```python
def apply(func, x):
    return func(x)

# Call the apply function with a lambda function
result = apply(lambda x: x**2, 3)
print(result)  # Output: 9
```

HIGHER-ORDER FUNCTIONS WITH LAMBDA

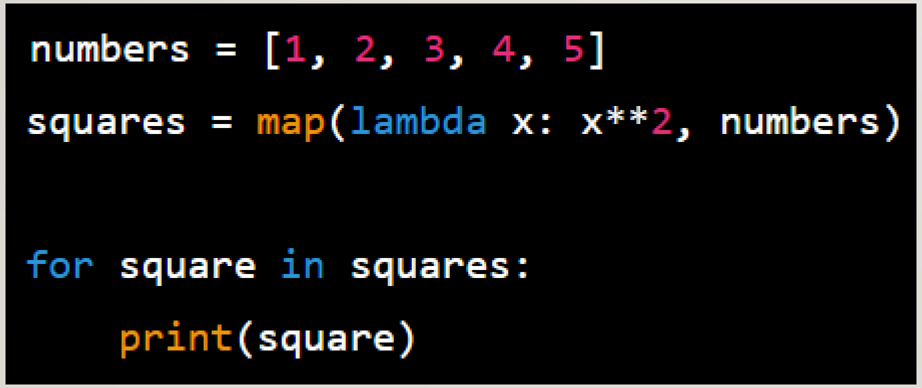
- In this example, we define a higher-order function called **make_adder(n)** that takes an argument n and returns a lambda function that adds n to its argument. We then call the **make_adder(n)** function with an argument of 3, which creates a lambda function that adds 3 to its argument. Finally, we call the lambda function with an argument of 4, which returns the result of 4 + 3 = 7.



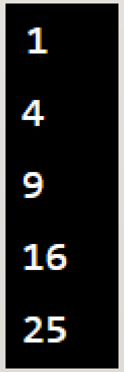
```
def make_adder(n):
    return lambda x: x + n

# Call the make_adder function to create a lambda function
add_3 = make_adder(3)
result = add_3(4)
print(result)  # Output: 7
```

HIGHER-ORDER FUNCTIONS USING BUILT-IN FUNCTIONS AND LAMBDA

- The **map()** function is a built-in higher-order function in Python that applies a function to each element of an iterable and returns a new iterable of the results.
- Here's an example of using **map()** with a lambda expression to compute the square of each number in a list:



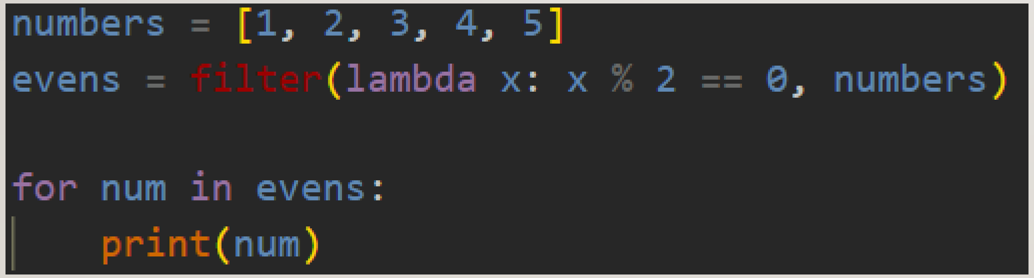
```python
numbers = [1, 2, 3, 4, 5]
squares = map(lambda x: x**2, numbers)

for square in squares:
    print(square)
```



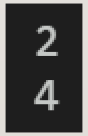
```
1
4
9
16
25
```

HIGHER-ORDER FUNCTIONS USING BUILT-IN FUNCTIONS AND LAMBDA

- The **filter()** function is another built-in higher-order function in Python that takes a function and an iterable as input and returns a new iterable containing only the elements of the original iterable for which the function returns True.
- Here's an example of using **filter()** with a lambda expression to filter a list of numbers to only include the even ones:



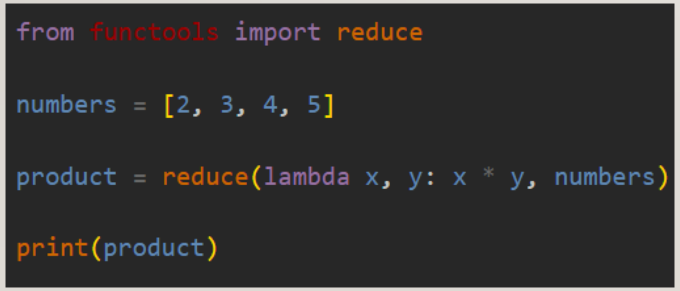
```python
numbers = [1, 2, 3, 4, 5]
evens = filter(lambda x: x % 2 == 0, numbers)

for num in evens:
    print(num)
```



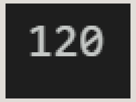
```
2
4
```

HIGHER-ORDER FUNCTIONS USING BUILT-IN FUNCTIONS AND LAMBDA

- In this example, the **reduce()** function is used to apply the lambda function to the list of numbers, producing a single result. The lambda function takes two arguments, x and y, and returns their product (x * y). The reduce() function starts by multiplying the first two numbers in the list (2 * 3 = 6), and then applies the lambda function to the result and the next number in the list (6 * 4 = 24, 24 * 5 = 120). The result (120) is then assigned to the variable product, which is printed to the console.
- Here's an example of how you might print the results of the computation to the console:



```
from functools import reduce

numbers = [2, 3, 4, 5]

product = reduce(lambda x, y: x * y, numbers)

print(product)
```

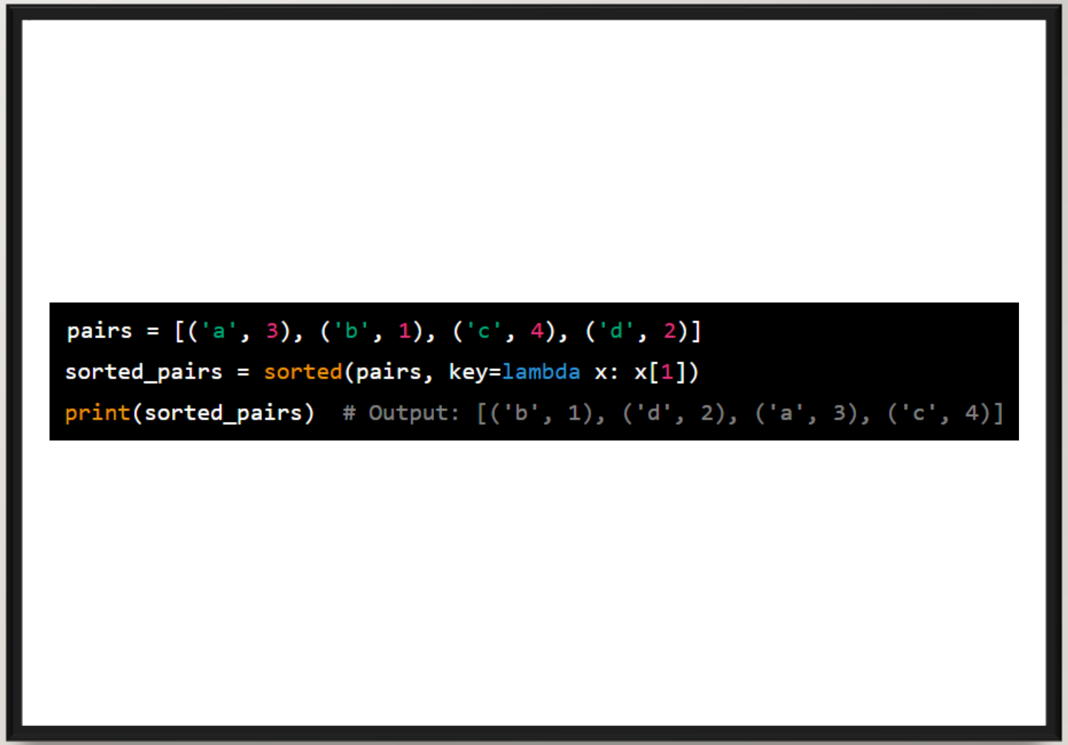


```
120
```

HIGHER-ORDER FUNCTIONS USING BUILT-IN FUNCTIONS AND LAMBDA

- In this example, the key argument is set to a lambda function that returns the second element of each tuple, which is used to determine the sorting order of the tuples.

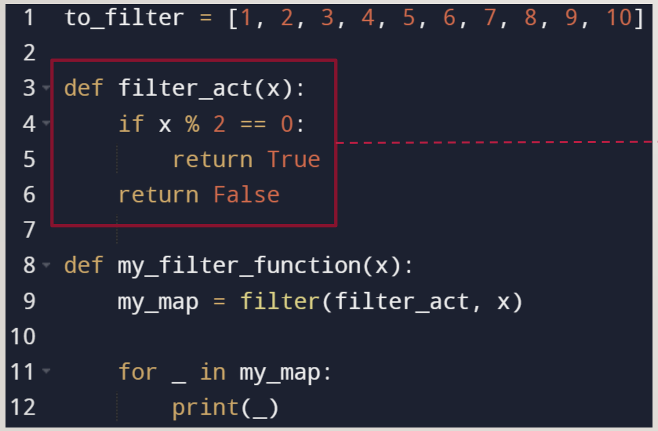
- Note that sorted() returns a new sorted list and does not modify the original iterable.



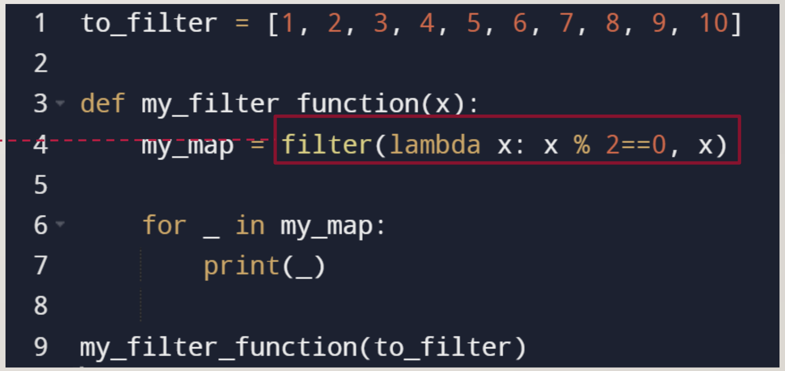
```
pairs = [('a', 3), ('b', 1), ('c', 4), ('d', 2)]
sorted_pairs = sorted(pairs, key=lambda x: x[1])
print(sorted_pairs)  # Output: [('b', 1), ('d', 2), ('a', 3), ('c', 4)]
```

WHY IS IT BETTER TO USE LAMBDA WITH HIGHER-ORDER BUILT-IN FUNCTIONS SOMETIMES

- It does not consume additional namespace for just a simple and very specific task
- Lessen the number of lines

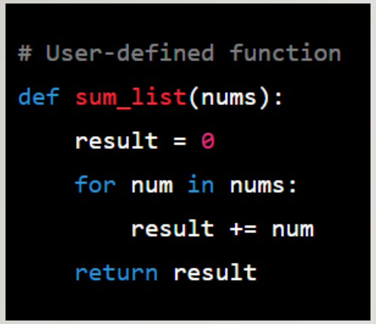


```python
to_filter = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

def filter_act(x):
    if x % 2 == 0:
        return True
    return False

def my_filter_function(x):
    my_map = filter(filter_act, x)

    for _ in my_map:
        print(_)
```

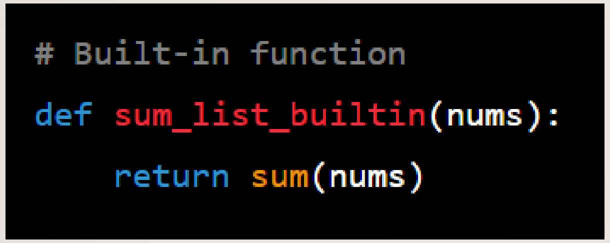


```python
to_filter = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

def my_filter_function(x):
    my_map = filter(lambda x: x % 2==0, x)

    for _ in my_map:
        print(_)

my_filter_function(to_filter)
```

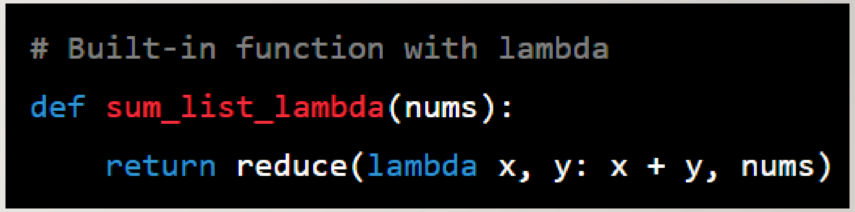
COMPARISONS



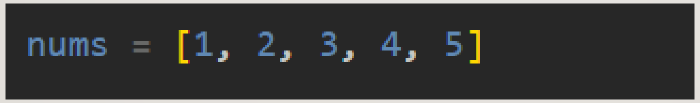
```python
# User-defined function
def sum_list(nums):
    result = 0
    for num in nums:
        result += num
    return result
```



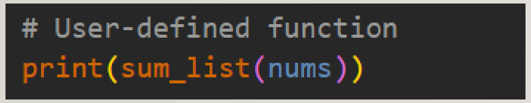
```python
# Built-in function
def sum_list_builtin(nums):
    return sum(nums)
```



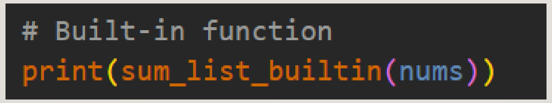
```python
# Built-in function with lambda
def sum_list_lambda(nums):
    return reduce(lambda x, y: x + y, nums)
```



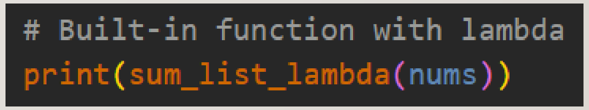
```python
nums = [1, 2, 3, 4, 5]
```



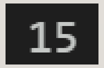
```python
# User-defined function
print(sum_list(nums))
```



```python
# Built-in function
print(sum_list_builtin(nums))
```



```python
# Built-in function with lambda
print(sum_list_lambda(nums))
```



```
15
```

MORE EXAMPLES

To access more examples with explanations, you may visit this link.

https://github.com/francismontalbo/learning_python