Python History Python was developed and introduced in the late 1980s by a Dutch programmer Guido Van Rossum . It is an object-oriented, interpreted, high-level programming language with high portability i.e. Python code/applications can be run on multiple platforms that have a version of Python installed. The name Python was also adapted from the name of a popular show “Monty Python” which Guido was watching at the time he developed the language.

Python data types: Numbers (i.e. integers, floats, complex etc.) Strings, Lists, Tuples, Dictionaries, Booleans, Print formatting

Numbers:

```
import math

#Integers: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
#Operators: +, -, *, /, **
#Parentheses: (, )
#Example: 5 + 3 * (2 - 1) / 4
addition = 5 + 3
subtraction = 5 - 3
multiplication = 5 * 3
division = 5 / 3
exponentiation = 5 ** 3
floor_division = 5 // 3
modulus = 5 % 3
exponential_base = (7-2*2)**3 / (12-3**2)

print(f"Addition: {addition}\nSubtraction: {subtraction}\
nMultiplication: {multiplication}\nDivision: {division}\
nExponentiation: {exponentiation}\nFloor Division: {floor_division}\
nModulus: {modulus}\nExponential Base: {exponential_base}")

#Floats are numbers with decimal points
#Floats: 1.0, 2.5, 3.14, 4.0, 5.75
float_addition = 5.5 + 3.2
float_subtraction = 5.5 - 3.2
float_multiplication = 5.5 * 3.2
float_division = 5.5 / 3.2
float_exponentiation = 5.5 ** 3.2
float_floor_division = 5.5 // 3.2
float_modulus = 5.5 % 3.2

print(f"Float Addition: {float_addition}\nFloat Subtraction:
{float_subtraction}\nFloat Multiplication: {float_multiplication}\
nFloat Division: {float_division}\nFloat Exponentiation:
{float_exponentiation}\nFloat Floor Division: {float_floor_division}\
nFloat Modulus: {float_modulus}")
# Complex numbers
# Complex numbers: 1+2j, 3-4j, 5+6j
complex_addition = (1+2j) + (3-4j)
complex_subtraction = (1+2j) - (3-4j)
```

```
complex_multiplication = (1+2j) * (3-4j)
complex_division = (1+2j) / (3-4j)
complex_exponentiation = (1+2j) ** 2

print(f"Complex Addition: {complex_addition}\nComplex Subtraction:
{complex_subtraction}\nComplex Multiplication:
{complex_multiplication}\nComplex Division: {complex_division}\
nComplex Exponentiation: {complex_exponentiation}")



# Exploring functions for int and float

# Absolute value
abs_value = abs(-5)

# Convert to binary
binary_value = bin(5)

# Convert to hexadecimal
hex_value = hex(255)

# Convert to octal
octal_value = oct(8)

# Power function
power_value = pow(2, 3)

# Round function
rounded_value = round(5.678, 2)

# Minimum and maximum
min_value = min(5, 3, 9, 1)
max_value = max(5, 3, 9, 1)

# Sum of elements
sum_value = sum([1, 2, 3, 4, 5])

# Floating point functions

# Square root
sqrt_value = math.sqrt(16)

# Exponential
exp_value = math.exp(2)

# Logarithm
log_value = math.log(10)

# Sine, Cosine, and Tangent
sin_value = math.sin(math.pi / 2)
```

```
cos_value = math.cos(math.pi)
tan_value = math.tan(math.pi / 4)

print(f"Absolute Value: {abs_value}\nBinary Value: {binary_value}\
nHexadecimal Value: {hex_value}\nOctal Value: {octal_value}\nPower
Value: {power_value}\nRounded Value: {rounded_value}\nMinimum Value:
{min_value}\nMaximum Value: {max_value}\nSum Value: {sum_value}\
nSquare Root: {sqrt_value}\nExponential: {exp_value}\nLogarithm:
{log_value}\nSine: {sin_value}\nCosine: {cos_value}\nTangent:
{tan_value}")
```

Strings

Strings in Python are sequences of characters enclosed in quotes. They can be created using single quotes (`'`), double quotes (`"`), or triple quotes (`'''` or `"""`). Strings are immutable, meaning their contents cannot be changed after creation. They support various operations such as concatenation, slicing, and formatting.

Key Points:

- Defined using single, double, or triple quotes.
- Immutable (cannot be changed after creation).
- Support concatenation, slicing, and formatting.
- Can include escape characters (e.g., `\n` for newline, `\t` for tab).

```
# Single quotes
single_quote_str = 'Michael Foster'

# Double quotes
double_quote_str = "David Beckham"

# Triple quotes
triple_quote_str = """Guillermo Giovani"""

# Correct usage with double quotes
double_quote_correct = "Hello Mark, it's nice to meet you!"

# String concatenation
first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name

# String replication
repeated_str = "I love Python programming, it's awesome\n" * 3

print(single_quote_str + "\n" + double_quote_str + "\n" +
triple_quote_str + "\n" + double_quote_correct + "\n" + full_name + "\
n" + repeated_str)

# String interpolation
```

```ruby
interpolated_str = "My name is #{first_name} #{last_name} and I love
Python programming."

# String length
length_of_full_name = full_name.length

# String methods
upper_case_str = full_name.upcase
lower_case_str = full_name.downcase
capitalized_str = full_name.capitalize
reversed_str = full_name.reverse

print(interpolated_str)
print(f"Length of full name: {length_of_full_name}")
print(f"Upper case: {upper_case_str}")
print(f"Lower case: {lower_case_str}")
print(f"Capitalized: {capitalized_str}")
print(f"Reversed: {reversed_str}")
# Check if string includes a substring
includes_substring = full_name.include?("John")

# Split string
split_str = full_name.split

# Strip whitespace
strip_str = "   John Doe   ".strip

# Replace substring
replaced_str = full_name.gsub("John", "Jane")

# Convert to array of characters
chars_array = full_name.chars

# Get substring
substring = full_name[0, 4]

# Count occurrences of a character
count_char = full_name.count("o")

# Check if string is empty
is_empty = full_name.empty?

# Check if string starts with a substring
starts_with = full_name.start_with?("John")

# Check if string ends with a substring
ends_with = full_name.end_with?("Doe")

print(f"Includes 'John': {includes_substring}, Split string:
{split_str}, Strip whitespace: '{strip_str}', Replace 'John' with
```

```
'Jane': {replaced_str}, Array of characters: {chars_array}, Substring
(first 4 chars): {substring}, Count occurrences of 'o': {count_char},
Is string empty: {is_empty}, Starts with 'John': {starts_with}, Ends
with 'Doe': {ends_with}")



# Slicing examples

# Get first character
first_char = full_name[0]

# Get last character
last_char = full_name[-1]

# Get first three characters
first_three_chars = full_name[0, 3]

# Get last three characters
last_three_chars = full_name[-3, 3]

# Get substring from index 1 to 4
substring_1_to_4 = full_name[1, 4]

# Get substring from index 2 to end
substring_from_2 = full_name[2..-1]

print(f"First character: {first_char}, Last character: {last_char},
First three characters: {first_three_chars}, Last three characters:
{last_three_chars}, Substring from index 1 to 4: {substring_1_to_4},
Substring from index 2 to end: {substring_from_2}")
```

Variables

```
# Program 1: Welcome three friends to a restaurant
# Taking input from the user for three friends' names
friend1 = input("Enter the name of first friend: ")
friend2 = input("Enter the name of second friend: ")
friend3 = input("Enter the name of third friend: ")

# Displaying a welcome message
print("\nWelcome to McDonald's, " + friend1 + ", " + friend2 + ", and
" + friend3 + "!")

# Program 2: Convert an integer to a floating-point number
# Taking an integer input from the user
num = int(input("\nEnter an integer: "))

# Converting integer to float
float_num = float(num)
```

```
# Displaying the result
print("The floating-point equivalent is:", float_num)

# Displaying the type
print("Type of float_num:", type(float_num))

# Program 3: Convert Celsius to Fahrenheit
# Taking input for temperature in Celsius
celsius = float(input("\nEnter temperature in Celsius: "))

# Formula to convert Celsius to Fahrenheit
fahrenheit = (celsius * 9/5) + 32

# Displaying the result
print("The temperature in Fahrenheit is:", fahrenheit)

# Displaying the type
print("Type of fahrenheit:", type(fahrenheit))
```

List

Lists are ordered collections of items that are mutable, meaning they can be changed after their creation. They can contain elements of different data types, including other lists. Lists are defined using square brackets, with elements separated by commas. They support various operations such as indexing, slicing, appending, and removing elements.

```
# List of integers
int_list = [1, 2, 3, 4, 6]
print("Integer List:", int_list)
print("Type:", type(int_list))

# List of floats
float_list = [12.1, 13.42, 15.6, 18.93, 20.0]
print("Float List:", float_list)
print("Type:", type(float_list))

# List of strings
string_list = ['New', 'Taken', 'Extra']
print("String List:", string_list)
print("Type:", type(string_list))

# List of boolean values
bool_list = [True, False]
print("Boolean List:", bool_list)
print("Type:", type(bool_list))

# List containing multiple data types
mixed_list = ['Derek', 25, 125.50, True]
print("Mixed Data List:", mixed_list)
```

```
print("Type:", type(mixed_list))

# Indexing in Lists
random_list = [1, 2, 3, 4, 'food']
third_element = random_list[2]  # Accessing the third element (index
2)
print("Third Element:", third_element)

# Nested Lists
random_list2 = [[1, 2, 3, 4, 'integers'], 'food', [12.3, 5.2, 10.0,
'floats']]
nested_value = random_list2[2][2]  # Accessing 10.0 from nested list
print("Nested Value:", nested_value)

# User Input using eval() to get a list
user_data = eval(input("Enter your info as ['Name', Age, Height,
Married]: "))
name, age, height, married = user_data  # Unpacking values
print("""Here are your details:
Name: {}
Age: {}
Height: {}m
Married: {}""".format(name, age, height, married))

# Example of list slicing
animals = ['Cat', 'Dog', 'Goat', 'Jaguar', 'Lion']
domestic_animals = animals[:3]  # Slicing first three elements
print("Domestic Animals:", domestic_animals)

# Example 1: Using .format() to print formatted output
name = "Saikat"
age = 25
balance = 5000

# Using .format() method to insert variables into a string
print("My name is {}, I am {} years old, and my bank balance is $
{}.".format(name, age, balance))

# Example 2: List slicing (extracting part of a list)
animals = ['Cat', 'Dog', 'Goat', 'Jaguar', 'Lion']

# Extracting first three elements (index 0 to 2, because slicing goes
up to end_index - 1)
domestic_animals = animals[:3]
print(domestic_animals)  # Output: ['Cat', 'Dog', 'Goat']
# Example 3: Nested list indexing (Accessing elements from sublists)
food_list = ['rice', 'salad', ['cake', 'ice-cream', 'cookies',
'doughnuts'], 'beans']

# Extracting specific treats from the nested list
```

```
treats = food_list[2][1:4]  # Accessing index 2 (sublist), then
slicing from index 1 to 3
print("I love {}, {}, and {}.".format(treats[0], treats[1],
treats[2]))

# Example 4: String Indexing (Extracting a substring)
word = "Exceptional"

# Extracting "Except" from "Exceptional"
new_string = word[:6]  # Slicing from index 0 to 5
print(new_string)  # Output: Except

# Example 5: Changing an element in a list (List Mutability)
fruits = ['Apple', 'Orange', 'Banana', 'Cashew', 'Almond']

# Replacing 'Orange' with 'Guava'
fruits[1] = 'Guava'
print(fruits)  # Output: ['Apple', 'Guava', 'Banana', 'Cashew',
'Almond']

# Example 6: Changing an element in a nested list
nested_list = ['Apple', ['Orange', 'Guava'], 'Banana']

# Replacing 'Orange' with 'Mango'
nested_list[1][0] = 'Mango'
print(nested_list)  # Output: ['Apple', ['Mango', 'Guava'], 'Banana']

# Example 7: Appending elements to a list
fruits.append('Pawpaw')  # Adding 'Pawpaw' at the end
print(fruits)  # Output: ['Apple', 'Guava', 'Banana', 'Cashew',
'Almond', 'Pawpaw']

# Example 8: Appending a sublist to a list
foods = ['Beans', 'Plantain', 'Fish']
foods.append(['Rice', 'Wheat'])  # Adding a sublist
print(foods)  # Output: ['Beans', 'Plantain', 'Fish', ['Rice',
'Wheat']]

# Example 9: Using .extend() to add multiple elements to a list
foods.extend(['Bread', 'Eggs'])  # Adds elements separately (not as a
sublist)
print(foods)  # Output: ['Beans', 'Plantain', 'Fish', ['Rice',
'Wheat'], 'Bread', 'Eggs']

# Example 10: Inserting elements at a specific index
numbers = [1, 2, 4, 5, 6, 7, 8, 10]

# Inserting missing numbers 3 and 9
numbers.insert(2, 3)  # Insert 3 at index 2
numbers.insert(8, 9)  # Insert 9 at index 8
```

```
print(numbers)  # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Example 11: Removing an element from a list
numbers.append(11)  # Adding 11
print(numbers)  # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

numbers.remove(11)  # Removing 11
print(numbers)  # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Example 12: List Addition and Multiplication
list1 = ['A', 'B', 'C', 'D']
list2 = [1, 2, 3, 4]

# Adding two lists together
combined_list = list1 + list2
print(combined_list)  # Output: ['A', 'B', 'C', 'D', 1, 2, 3, 4]

# Multiplying a list (Repeating elements)
print(list2 * 2)  # Output: [1, 2, 3, 4, 1, 2, 3, 4]
```

Tuples

Tuples are similar to lists as they are a sequence of comma-separated values. However, unlike lists, tuples are immutable, meaning they cannot be changed after creation. Tuples are defined using parentheses ().

Key Points:

- Tuples are immutable.
- Defined using parentheses ().
- Data can be accessed using indexing.

Use Case: Tuples are useful when you need to ensure that data cannot be modified. For example, in a banking application, account details that should not be changed can be stored in a tuple. Users can retrieve this data, but they cannot modify it.

```
# Example 24: Declaring Tuples

# Tuples can be declared in two ways:

# Way 1: Using parentheses (traditional and recommended method)
my_tuple = (1, 2, 3, 4, 5)
print(my_tuple)  # Output: (1, 2, 3, 4, 5)
print(type(my_tuple))  # Output: <class 'tuple'>

# Way 2: Using tuple packing (without parentheses)
my_tuple2 = 1, 2, 3, 4, 5
print(my_tuple2)  # Output: (1, 2, 3, 4, 5)
print(type(my_tuple2))  # Output: <class 'tuple'>
```

```python
# Any sequence of comma-separated items gets assigned as a tuple
automatically.

# Example 25: Comparing Lists with Tuples (Memory Efficiency)

# Defining a list and a tuple with the same elements
my_list = ['Men', 'Index', 1, 2, 3]
my_tuple = ('Men', 'Index', 1, 2, 3)

# Getting the memory size of list and tuple
import sys
list_size = sys.getsizeof(my_list)
tuple_size = sys.getsizeof(my_tuple)

print(f'Size of list is {list_size}, and size of tuple is
{tuple_size}.')
# Example Output: Size of list is 96, and size of tuple is 80.

# Tuples take less memory than lists, making them more efficient in
resource-limited environments.

# Example 26: Grabbing Elements from a Tuple

# Accessing elements using indexing
A = my_tuple[0]  # 'Men'
B = my_tuple[1]  # 'Index'

print(f'The {B} of {A} is 0.')
# Output: The Index of Men is 0.

# Trying to change a tuple element (This will cause an error)
# my_tuple[0] = 'Women'  # TypeError: 'tuple' object does not support
item assignment

# Example 27: Tuple Packing and Unpacking

# Tuple Packing (Assigning multiple values to a single tuple)
Tuple = 'Daniel', 'Dean', 'James'  # No parentheses needed

# Tuple Unpacking (Assigning values from tuple to variables)
Name1, Name2, Name3 = Tuple

print(Name1)  # Output: Daniel
print(Name2)  # Output: Dean
print(Name3)  # Output: James

# Tuple packing and unpacking allow convenient variable assignment.
# Example 28: Collecting Data with Tuples

# Defining a nested tuple to store user information
user_info = (('Name', 'James'), ('Age', 22))  # Nested tuple
```

```python
# Unpacking inner tuples
A, B = user_info[0]  # Unpacking ('Name', 'James')
C, D = user_info[1]  # Unpacking ('Age', 22)

# Printing user information
print(A, ':', B)  # Output: Name : James
print(C, ':', D)  # Output: Age : 22
```

Comparison between Tuple and List in Python

Feature	Tuple (tuple)	List (list)
Mutability	Immutable (cannot be changed after creation)	Mutable (can be modified: add, remove, update elements)
Syntax	Defined using `()` (e.g., `t = (1, 2, 3)`)	Defined using `[]` (e.g., `l = [1, 2, 3]`)
Performance	Faster (requires less memory)	Slower (more memory used due to dynamic resizing)
Memory Usage	Uses less memory	Uses more memory
Operations	Supports indexing, slicing, iteration	Supports all tuple operations + modifications (append, remove, sort, etc.)
Use Case	Best for fixed data (e.g., coordinates, configurations)	Best for dynamic data (e.g., lists of users, items, etc.)
Safety	More secure (immutable, prevents accidental modification)	Less secure (mutable, can be changed anytime)
Iteration Speed	Faster iteration due to immutability	Slower iteration due to dynamic nature

Tuple Example (Immutable)

```python t = (1, 2, 3) print(t[0]) # Output: 1

# t[0] = 10 # ❌ TypeError: 'tuple' object does not support item assignment

l = [1, 2, 3] l[0] = 10 # ✅ Allowed print(l) # Output: [10, 2, 3]

## Dictonaries in Python

Dictionaries in Python are unordered collections of items. They store data in key-value pairs, where each key is unique and maps to a value. Dictionaries are defined using curly braces `{}`,
```

with keys and values separated by a colon `:`. They are mutable, meaning their contents can be changed after creation.

Key Points:

- Unordered collection of key-value pairs.
- Keys must be unique and immutable (e.g., strings, numbers, tuples).
- Values can be of any data type and can be duplicated.
- Defined using curly braces `{}`.
- Accessed using keys.

Use Case: Dictionaries are useful for storing data that needs to be quickly retrieved using a unique key. For example, they can be used to store user information, where the username is the key and the user details are the values.

```
# Dictionary declaration
my_Dict = {
    'Key1': 'Movies',  # A string as key and value is a string
    'Key2': ['Iron Man', 'Avengers']  # A string as key and value is a
list
}

# Print the dictionary and check its type
print(my_Dict)
print(type(my_Dict))  # Output: <class 'dict'>

# Grabbing dictionary elements by their keys
A = my_Dict['Key1']  # Accessing the value for 'Key1'
B = my_Dict['Key2']  # Accessing the value for 'Key2'

# Print the grabbed elements
print(A, ':', B)  # Output: Movies : ['Iron Man', 'Avengers']

# Example of a nested dictionary
Acct_Dict = {
    'Name': 'Customer1',  # Key: 'Name', Value: 'Customer1'
    'Account type': {  # Key: 'Account type', Value: another
dictionary
        'type1': 'Savings',  # Key: 'type1', Value: 'Savings'
        'type2': 'Current'  # Key: 'type2', Value: 'Current'
    }
}

# Printing a value from a nested dictionary
print('You have a ' + Acct_Dict['Account type']['type1'] + ' account')
# Output: You have a Savings account

# Checking the keys of a dictionary using the .keys() method
print(Acct_Dict.keys())  # Output: dict_keys(['Name', 'Account type'])
```

```
# Checking the keys of the inner dictionary
print(Acct_Dict['Account type'].keys())  # Output: dict_keys(['type1',
'type2'])

# Exploring available methods for different data types using the dir()
function

# Declare some datatypes for testing
String = 'Bob'
integer = 100
Float = 25.3
List = ['Man']
Tuple = 5,  # Tuple with a single value, needs a comma
Dictionary = dict([('Name', 'Max')])  # Creating a dictionary using
the dict() function

# Check the methods available for each datatype
A = dir(String)
B = dir(integer)
C = dir(Float)
D = dir(List)
E = dir(Tuple)
F = dir(Dictionary)

# Print the available methods for each datatype
print('''Here are the methods possible with each type:
Strings
{}
Integers
{}
Floats
{}
Lists
{}
Tuples
{}
Dictionaries
{}
'''.format(A, B, C, D, E, F))

# Notes:
# - The dir() function returns a list of valid attributes and methods
for an object.
# - The semicolon can be used to write multiple commands in one line
(although not recommended for clarity).
```

Booleans in Python

Booleans are conditional datatypes used to determine the state of a statement or block of code. They have two possible values: `True` and `False`, which are equivalent to the integer values `1` and `0`, respectively.

Example of Boolean values:

```python A = True # Boolean value 'True' print(type(A)) # Output: <class 'bool'>

Integer Value of Boolean You can convert a Boolean to an integer using the int() function. The corresponding integer value for True is 1 and for False is 0. print(int(A)) # Output: 1 (Since A is True)

## Comparison Operators

Comparison operators are used to check for the validity of a comparison between two values. The following are the comparison operators in Python:

| Operator | Description |
|---|---|
| `<` | Less than |
| `>` | Greater than |
| `==` | Equal to |
| `<=` | Less than or equal to |
| `>=` | Greater than or equal to |
| `!=` | Not equal to (Not operator) |

These operators return **Boolean values** (`True` or `False`) based on the comparison.

## Example 32: Checking Conditions Using Comparison Operators

The following code demonstrates the use of comparison operators to compare values:

```python

# This code illustrates the Boolean comparison output

print(5 < 10) # Output: True (5 is less than 10) print(3 > 4) # Output: False (3 is not greater than 4) print('Bob' == 'Mary') # Output: False (Bob is not equal to Mary) print(True == 1) # Output: True (True is equal to 1) print(False == 0) # Output: True (False is equal to 0) print(True != 1) # Output: False (True is equal to 1)

# The ‘in’ operator checks for the existence of an element/value

print('Max' in 'Max Payne') # Output: True (Max is in 'Max Payne') print(2 in [1, 3, 4, 5]) # Output: False (2 is not in the list [1, 3, 4, 5]) print(True in [1, 0]) # Output: True (True is in the list [1, 0])

## Logical Operators in Python

Logical operators are the Python equivalents of logic gates and are used to perform logical operations. They follow basic Boolean algebra rules and can be combined using De Morgan's Law.

De Morgan's Law simplifies Boolean logic using operations originally invented by George Boole. The logical operators in Python are:

| Operator | Description |
|---|---|
| `and` | Evaluates to `True` if, and only if, both operands are `True`, otherwise `False` |
| `or` | Evaluates to `True` if at least one operand is `True`, otherwise `False` |
| `not` | Inverts the value/operation of its operand |

## Example 33: Logical Operation with the `and` Operator

The `and` operator returns `True` only if both operands are `True`. Otherwise, it returns `False`.

```python

# The 'and' operator

print(True & True) # Output: True print(True and False) # Output: False print(False and False) # Output: False print(False and True) # Output: False print('foo' in 'foobar' and 1 < 2) # Output: True

## Conditional Statements and Loops in Python

```python
# Conditional Statements and Loops

# Conditional statements are used to execute certain blocks of code based on conditions.


# Example 35: Grant access if the user enters the correct password
password_pool = ('Smithcrete', 'Alex@456', 'CEO4life')  # Tuple of valid passwords
user_password = input('Please enter your password: \t')  # Get password from user
```
```

```
# IF condition: Check if the user password is in the password pool
if user_password in password_pool:
    print('\n Access granted!')  # If the password is correct, grant
access
else:
    print('\n Access Denied! Calling Security …')  # If the password
is incorrect, raise an alarm



# Example 36: Movie rent-overdue price calculation

price1 = 5  # $5 for each day past due within the first 3 days
price2 = 7  # $7 for each day past due after 3 days

days_past_due = eval(input('How many days past due:\t'))  # Get number
of days the movie is overdue

# If statement: Check if the days overdue are less than or equal to 3
if days_past_due <= 3:
    print('\nYou owe $', (days_past_due * price1))  # Calculate the
amount owed for the first 3 days
else:
    print('\nYou owe $', (3 * price1 + (days_past_due - 3) * price2))
# Calculate amount owed if overdue for more than 3 days



# Example 37: Grant access or raise alarm with IF-ELSE
password_pool = ('Smithcrete', 'Alex@456', 'CEO4life')  # Tuple of
valid passwords
user_password = input('Please enter your password: \t')  # Get
password from user

# IF-ELSE conditions: Check if the password is valid
if user_password in password_pool:
    print('\n Access granted!')  # If password is valid, grant access
else:
    print('\n Access Denied! Calling Security …')  # If password is
invalid, raise an alarm



# Example 38: Improved password check with IF-ELIF-ELSE

password_pool = ('Smithcrete', 'Alex@456', 'CEO4life')  # Tuple of
valid passwords
user_password = input('Please enter your password: \t')  # Get
password from user

# IF-ELIF-ELSE conditions: Check multiple conditions
```

```
if user_password in password_pool and user_password == 'Smithcrete':
    print('\nAccess granted! Welcome Dr. Smith')  # Special message
for Smithcrete password
elif user_password in password_pool and user_password == 'Alex@456':
    print('\nAccess granted! Welcome Mr. Alexander')  # Special
message for Alex@456 password
elif user_password in password_pool and user_password == 'CEO4life':
    print('\nAccess granted! Welcome Mr. CEO')  # Special message for
CEO4life password
else:
    print('\nAccess Denied! Calling Security …')  # Raise alarm if
password is incorrect



# Example Exercise: Check if a word exists in both lists using logical
operators

list1 = ['Beans', 'Wheat', 'Bread']
list2 = ['Rice', 'Plantain', 'Pizza', 'Spaghetti']

# a. Check if 'Rice' exists in both lists
if 'Rice' in list1 and 'Rice' in list2:
    print("Rice is in both lists")

# b. Check if 'Pizza' exists in at least one of the lists
if 'Pizza' in list1 or 'Pizza' in list2:
    print("Pizza is in at least one list")
```

WHILE Loop

The `while` loop is used to execute a set of statements or code as long as a specified condition is true. The conditional statement that controls the algorithm is called a flag and is always true for all non-zero values. When the flag becomes zero, the `while` loop then passes on to the next line of code following it. There could be a single statement or multiple within the `while` loop, and Python supports the `else` statement for the `while` loop.

```
# WHILE loop Example 39: Writing a while loop to print a statement 5
times
i = 1  # counter variable
while i < 6:  # flag: loop will run as long as i is less than 6
    print(i, ': I love Python')  # print the current iteration number
with the statement
    i = i + 1  # increment the counter by 1 after each iteration
else:
    print('\nThe program has completed')  # once the loop condition is
false, print completion message

# Example 40: Using the continue statement to skip a number (skipping
```

```
8)
i = 0  # counter variable
while i <= 9:  # loop runs as long as i is less than or equal to 9
    i = i + 1  # increment counter
    if i == 8:  # if counter equals 8, skip this iteration
        continue  # skip the current iteration and move to the next
one
    print(i)  # print the current value of i
else:
    print('\nThe program has ended')  # once the loop ends, print
completion message

# Example 41: Indefinite loop with break and continue
i = 1  # loop variable for counting numbers
j = 1  # loop variable for tracking the number of iterations
while i <= 10:  # loop will continue until i reaches 10
    print('Iteration', j)  # print the current iteration number
    if j >= 15:  # if iteration count reaches 15, break the loop
        break
    j = j + 1  # increment iteration count
    if i == 8:  # when i equals 8, skip the current iteration
        continue
    print('\tvalue =', i)  # print the current value of i
    i = i + 1  # increment i by 1 for each iteration
else:
    print('\nThe program has ended')  # when the loop ends, print
completion message
```

For Loop

For loops are used for iterating through a sequence of values. They execute a statement of code for each element present in the target sequence. Like while loops, they can also run indefinitely depending on how they are declared.

Key Points:

- Used for iterating over a sequence (e.g., list, tuple, dictionary, set, string).
- Executes a block of code for each element in the sequence.
- Can be controlled using `break` and `continue` statements.

```
# Example 1: Iterating over a list
fruits = ['Apple', 'Banana', 'Cherry']
for fruit in fruits:
    print(fruit)

# Example 2: Iterating over a string
word = "Python"
for letter in word:
    print(letter)
```

```python
# Example 3: Iterating over a range of numbers
for num in range(5):  # range(5) generates numbers from 0 to 4
    print(num)

# Example 4: Using range with start and end values
for num in range(2, 10):  # range(2, 10) generates numbers from 2 to 9
    print(num)

# Example 5: Using range with step value
for num in range(1, 10, 2):  # range(1, 10, 2) generates numbers from
1 to 9 with a step of 2
    print(num)

# Example 6: Iterating over a dictionary
person = {'Name': 'Alice', 'Age': 25, 'City': 'New York'}
for key in person:
    print(key, person[key])

# Example 7: Iterating over dictionary items
for key, value in person.items():
    print(key, value)

# Example 8: Nested for loop
for i in range(3):
    for j in range(2):
        print(f"i: {i}, j: {j}")

# Example 9: Using break statement
for num in range(10):
    if num == 5:
        break
    print(num)

# Example 10: Using continue statement
for num in range(10):
    if num == 5:
        continue
    print(num)

# Example 11: Using else with for loop
for num in range(5):
    print(num)
else:
    print("Loop completed")

# Example 12: Iterating over a list with index
for index, fruit in enumerate(fruits):
    print(index, fruit)

# Example 13: Iterating over a tuple
```

```
colors = ('Red', 'Green', 'Blue')
for color in colors:
    print(color)

# Example 14: Iterating over a set
unique_numbers = {1, 2, 3, 4, 5}
for number in unique_numbers:
    print(number)
```

List Comprehension

```
# Example 44: List comprehension to square all elements in a list
List = list(range(10))  # Create a list with numbers from 0 to 9
squared_list = [item**2 for item in List]  # Square each item in the
list using list comprehension
# Example 45: Using list comprehension with an IF condition to grab
the first three letters of each string item in a list
List = ['Matrix', 'Trilogy', 1, 3.4, 'Cattle']  # Mixed list
containing strings and other types
new_list = [things[:3] for things in List if type(things) == str]  #
Grab the first three letters of strings only
# Exercise: Modify the list and grab the first three letters of words
that have at least three letters
List.extend(['Jeep', 'Man', 'Go'])  # Add words to the list using the
extend method
new_list = [things[:3] for things in List if type(things) == str and
len(things) >= 3]  # Grab first 3 letters only for words with 3 or
more letters
# Example 46: List comprehension with IF-ELSE to create a numeric list
with squares of odd numbers and halves of even numbers
nitems = eval(input('Please enter number of list items: '))  # Take
user input for the number of list items
new_list = [x**2 if x % 2 != 0 else x/2 for x in range(1, nitems + 1)]
# If x is odd, square it, else halve it
print('\nHere is your custom list\n', new_list)  # Print the resulting
list
```

Functions in Python

In Python, functions are essential for organizing code into reusable blocks. Rather than repeating code, we can define functions that execute specific tasks. Python also provides in-built functions like range(), list(), and type(), which are used frequently in programming.

However, Python also allows programmers to define custom functions to perform specialized tasks. This is done using the def keyword, which allows us to create functions that can be called multiple times with different inputs.

```
# Basic Functions
```

```python
# 1. Simple Function
def greet():
    print("Hello, World!")

greet()  # Output: Hello, World!

# 2. Function with Parameters
def greet_person(name):
    print(f"Hello, {name}!")

greet_person("Alice")  # Output: Hello, Alice!

# 3. Function with Return Value
def add(a, b):
    return a + b

result = add(5, 3)
print(result)  # Output: 8

# Intermediate Functions

# 4. Default Arguments
def greet_person(name="Guest"):
    print(f"Hello, {name}!")

greet_person()  # Output: Hello, Guest!
greet_person("Bob")  # Output: Hello, Bob!

# 5. Keyword Arguments
def describe_person(name, age, city):
    print(f"{name} is {age} years old and lives in {city}.")

describe_person(age=30, name="Charlie", city="New York")  # Output:
Charlie is 30 years old and lives in New York.

# 6. Variable-Length Arguments
def sum_all(*args):
    return sum(args)

print(sum_all(1, 2, 3, 4))  # Output: 10

# 7. Lambda Functions
square = lambda x: x ** 2
print(square(5))  # Output: 25

# Advanced Functions

# 8. Nested Functions
def outer_function(text):
    def inner_function():
        print(text)
```

```
    inner_function()

outer_function("Hello from nested function!")  # Output: Hello from
nested function!

# 9. Closures
def make_multiplier(x):
    def multiplier(n):
        return x * n
    return multiplier

times3 = make_multiplier(3)
print(times3(10))  # Output: 30

# 10. Decorators
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
# Output:
# Something is happening before the function is called.
# Hello!
# Something is happening after the function is called.

# 11. Generator Functions
def countdown(n):
    while n > 0:
        yield n
        n -= 1

for number in countdown(5):
    print(number)
# Output:
# 5
# 4
# 3
# 2
# 1
```

Map Function

The `map` function is another convenient way of working with lists, especially in cases where there is a need to pass the elements of a list to a function in an iteration. As expected, `for` loops are the first consideration for such a task. However, the `map` function, which iterates through every element in a list and passes them to a function argument, simplifies such tasks.

Example

```
# Example 1: Using map with a built-in function
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x ** 2, numbers))
print(squared_numbers)  # Output: [1, 4, 9, 16, 25]

# Example 2: Using map with a User-defined Function
def add_five(x):
    return x + 5

numbers = [1, 2, 3, 4, 5]
new_numbers = list(map(add_five, numbers))
print(new_numbers)  # Output: [6, 7, 8, 9, 10]

# Example 3: Using map with multiple iterables
numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]
sum_numbers = list(map(lambda x, y: x + y, numbers1, numbers2))
print(sum_numbers)  # Output: [5, 7, 9]

# Example 4: Using map with a string
chars = ['a', 'b', 'c', 'd']
upper_chars = list(map(str.upper, chars))
print(upper_chars)  # Output: ['A', 'B', 'C', 'D']

# Example 5: Using map with a dictionary
dict_values = {'a': 1, 'b': 2, 'c': 3}
keys = list(map(lambda x: x.upper(), dict_values.keys()))
values = list(map(lambda x: x * 2, dict_values.values()))
print(keys)    # Output: ['A', 'B', 'C']
print(values)  # Output: [2, 4, 6]

# Example 6: Using map with a tuple
numbers = (1, 2, 3, 4, 5)
squared_numbers = tuple(map(lambda x: x ** 2, numbers))
print(squared_numbers)  # Output: (1, 4, 9, 16, 25)

# Example 7: Using map with a set
numbers = {1, 2, 3, 4, 5}
squared_numbers = set(map(lambda x: x ** 2, numbers))
print(squared_numbers)  # Output: {1, 4, 9, 16, 25}

# Example 8: Using map with a list of tuples
```

```
points = [(1, 2), (3, 4), (5, 6)]
sum_points = list(map(lambda point: point[0] + point[1], points))
print(sum_points)  # Output: [3, 7, 11]

# Example 9: Using map with a list of dictionaries
students = [{'name': 'Alice', 'score': 85}, {'name': 'Bob', 'score':
90}]
names = list(map(lambda student: student['name'], students))
print(names)  # Output: ['Alice', 'Bob']

# Example 10: Using map with a custom class
class Student:
    def __init__(self, name, score):
        self.name = name
        self.score = score

students = [Student('Alice', 85), Student('Bob', 90)]
names = list(map(lambda student: student.name, students))
print(names)  # Output: ['Alice', 'Bob']
```

Filter Function

The `filter` function is used to filter elements from an iterable (e.g., list, tuple, set) based on a function that returns either `True` or `False`. The `filter` function constructs an iterator from elements of the iterable for which the function returns `True`.

Syntax

filter(function, iterable)

```


# Example 1: Filtering even numbers from a list
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Output: [2, 4, 6, 8, 10]

# Example 2: Filtering odd numbers from a list
odd_numbers = list(filter(lambda x: x % 2 != 0, numbers))
print(odd_numbers)  # Output: [1, 3, 5, 7, 9]

# Example 3: Filtering positive numbers from a list
numbers = [-10, -5, 0, 5, 10]
positive_numbers = list(filter(lambda x: x > 0, numbers))
print(positive_numbers)  # Output: [5, 10]

# Example 4: Filtering strings with length greater than 3
words = ["apple", "bat", "cat", "elephant", "dog"]
long_words = list(filter(lambda x: len(x) > 3, words))
print(long_words)  # Output: ['apple', 'elephant']
```

```python
# Example 5: Filtering vowels from a list of characters
chars = ['a', 'b', 'c', 'd', 'e', 'i', 'o', 'u']
vowels = list(filter(lambda x: x in 'aeiou', chars))
print(vowels)  # Output: ['a', 'e', 'i', 'o', 'u']

# Example 6: Filtering dictionary keys based on their values
dict_values = {'a': 1, 'b': 2, 'c': 3}
filtered_keys = list(filter(lambda k: dict_values[k] > 1,
dict_values))
print(filtered_keys)  # Output: ['b', 'c']

# Example 7: Filtering numbers greater than a threshold from a set
numbers = {1, 2, 3, 4, 5}
threshold = 3
filtered_numbers = set(filter(lambda x: x > threshold, numbers))
print(filtered_numbers)  # Output: {4, 5}

# Example 8: Filtering tuples based on the sum of their elements
points = [(1, 2), (3, 4), (5, 6)]
filtered_points = list(filter(lambda point: sum(point) > 5, points))
print(filtered_points)  # Output: [(3, 4), (5, 6)]

# Example 9: Filtering students with scores greater than 85
class Student:
    def __init__(self, name, score):
        self.name = name
        self.score = score

students = [Student('Alice', 85), Student('Bob', 90)]
high_scorers = list(filter(lambda student: student.score > 85,
students))
high_scorer_names = list(map(lambda student: student.name,
high_scorers))
print(high_scorer_names)  # Output: ['Bob']

# Example 10: Filtering elements from a list of mixed data types
mixed_list = ['Matrix', 'Trilogy', 1, 3.4, 'Cattle']
strings_only = list(filter(lambda x: isinstance(x, str), mixed_list))
print(strings_only)  # Output: ['Matrix', 'Trilogy', 'Cattle']

#    1.    .format() Method Exercise:
quantity = 'Gravity'
unit = 'm/s^2'
value = 10
print("{} has a value of {} {}".format(quantity, value, unit))
#    2.    Function to Check for Word “good” in a List:
def check_good(word_list):
    if 'good' in word_list:
        print(True)
    else:
```

```
        print(False)

# Test the function
words = ['bad', 'good', 'awesome']
check_good(words)
#    3.   Lambda Expression and filter() to Extract Words Not
Starting with ‘b’:
words = ['bread', 'rice', 'butter', 'beans', 'pizza', 'lasagna',
'eggs']
result = list(filter(lambda word: not word.startswith('b'), words))
print(result)
#    4.   Indexing to Grab “hello Python” from a Nested List:
lst = [1, 2, [3, 4], [5, [100, 200, ['hello Python']], 23, 11], 1, 7]
word = lst[3][1][2][0]  # Grabbing "hello Python"
print(word)
#    5.   Indexing to Grab “hello Python” from a Nested Dictionary:
d = {'k1': [1, 2, 3, {'tricky': ['oh', 'man', 'inception', {'target':
[1, 2, 3, 'hello']}]}]}
word = d['k1'][3]['tricky'][3]['target'][3]  # Grabbing "hello"
print(word)
#    6.   Function to Print Student Name and Grade Based on Score:
def student_grade(name, score):
    if score < 40:
        print(f"Hello {name}, you had an F")
    elif 40 <= score < 45:
        print(f"Hello {name}, you had an E")
    elif 45 <= score < 50:
        print(f"Hello {name}, you had a D")
    elif 50 <= score < 60:
        print(f"Hello {name}, you had a C")
    elif 60 <= score < 70:
        print(f"Hello {name}, you had a B")
    else:
        print(f"Hello {name}, you had an A, Cool!")

# Test the function
student_grade('Alice', 85)  # Output: Hello Alice, you had an A, Cool!
student_grade('Bob', 38)    # Output: Hello Bob, you had an F
```