IMPORTANT CONCEPTS-

Exception Handling - Exception handling in Python is a mechanism that allows you to handle and manage runtime errors or exceptional situations that may occur during the execution of a program. When an error or exception occurs, it disrupts the normal flow of the program, and without proper handling, it can cause the program to terminate abruptly.

Python Try Except

The `try` block lets you test a block of code for errors.

The `except` block lets you handle the error.

The `else` block lets you execute code when there is no error.

The `finally` block lets you execute code, regardless of the result of the try- and except blocks.

Exception Handling

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

These exceptions can be handled using the `try` statement:

Example

The `try` block will generate an exception, because `x` is not defined:

```
try:
  print(x)
```

```
except:
  print("An exception occurred")
```

Since the try block raises an error, the except block will be executed.

Without the try block, the program will crash and raise an error:

Example

This statement will raise an error, because x is not defined:

```
print(x)
```

Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

Example

Print one message if the try block raises a NameError and another for other errors:

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

ADVERTISEMENT

Else

You can use the else keyword to define a block of code to be executed if no errors were raised:

Example

In this example, the `try` block does not generate any error:

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

Finally

The `finally` block, if specified, will be executed regardless if the try block raises an error or not.

Example

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

This can be useful to close objects and clean up resources:

Example

Try to open and write to a file that is not writable:

```
try:
  f = open("demofile.txt")
  try:
    f.write("Lorum Ipsum")
  except:
    print("Something went wrong when writing to the file")
  finally:
    f.close()
except:
  print("Something went wrong when opening the file")
```

The program can continue, without leaving the file object open.

Raise an exception

As a Python developer you can choose to throw an exception if a condition occurs.

To throw (or raise) an exception, use the `raise` keyword.

Example

Raise an error and stop the program if x is lower than 0:

```
x = -1

if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

The `raise` keyword is used to raise an exception.

You can define what kind of error to raise, and the text to print to the user.

Example

Raise a TypeError if x is not an integer:

```
x = "hello"

if not type(x) is int:
  raise TypeError("Only integers are allowed")
```

Dictionary-

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered*, changeable and do not allow duplicates.

Dictionaries are written with curly brackets, and have keys and values:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

Dictionary Items

Dictionary items are ordered, changeable, and does not allow duplicates.

Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict["brand"])
```

When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change.

Unordered means that the items does not have a defined order, you cannot refer to an item by using an index.

(As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.)

Changeable

Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.

Duplicates Not Allowed

Dictionaries cannot have two items with the same key:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(thisdict)
```

The dict() Constructor

It is also possible to use the `dict()` constructor to make a dictionary.

```
thisdict = dict(name = "John", age = 36, country = "Norway")
print(thisdict)
```

LEVERAGING A DICTIONARY-

"leverage" means using something to its maximum advantage or exploiting its potential. When it comes to a dictionary in Python, you can leverage its features to achieve various tasks efficiently.

Here are a few ways you can leverage a dictionary in Python:

1. Efficient Lookup: Dictionaries provide fast lookup by key. If you have a large dataset and need to retrieve values based on specific keys, a dictionary can offer superior performance compared to other data structures.
2. Mapping and Associations: Dictionaries are commonly used for mapping one value to another or establishing associations between different entities. By leveraging dictionaries, you can easily create relationships and access corresponding values.
3. Grouping and Aggregation: Dictionaries allow you to group and aggregate data based on specific keys. You can use dictionary keys to categorize and organize information, making it easier to perform operations like counting occurrences, calculating totals, or summarizing data.
4. Configurations and Settings: Dictionaries are often used to store configurations or settings for a program. By leveraging a dictionary, you can conveniently access and modify these settings at runtime, making your code more flexible and configurable.

5. Efficient Data Structure: Dictionaries provide an efficient way to store and manage key-value pairs. By leveraging the built-in dictionary operations, you can add, modify, or delete elements with ease, leading to clean and concise code.

Remember, leveraging a dictionary involves utilizing its inherent properties and operations to solve specific problems or optimize code. By understanding and effectively utilizing dictionaries, you can write more efficient and expressive Python programs.

To leverage a dictionary in Python, you can use its key-value pairs to perform various operations and achieve specific tasks. Here are a few examples of how you can leverage a dictionary:

1. Efficient Lookup and Retrieval:
    - Use dictionary keys as identifiers to quickly retrieve corresponding values.
    - Leverage dictionary's `get()` method to retrieve values with a default fallback if the key is not present.
    - Utilize the `in` operator to efficiently check if a key exists in the dictionary.
2. Mapping and Transformation:
    - Use dictionaries to map one set of values to another, such as converting codes to meaningful labels.
    - Leverage dictionary comprehensions to transform or filter the dictionary into a new dictionary with specific criteria.
3. Grouping and Aggregation:
    - Leverage dictionaries to group and aggregate data based on specific keys.
    - Use dictionaries to count occurrences, calculate sums or averages, or perform other aggregations.
4. Configurations and Settings:
    - Store program configurations or settings in a dictionary for easy access and modification.
    - Leverage the dictionary's key-value structure to manage and update settings at runtime.
5. Efficient Data Structure:
    - Use dictionaries to efficiently store and manage data that requires key-based access.
    - Leverage dictionary methods like `update()` to merge dictionaries or `pop()` to remove and retrieve values based on keys.

6. Iteration and Transformation:
    - Iterate over dictionary keys, values, or key-value pairs to perform specific operations or transformations.
    - Leverage dictionary comprehensions or generator expressions to transform or filter the dictionary's elements.

Example-

```
# Example: Leverage a dictionary for student data

# Create a dictionary to store student information
student_data = {
    'John': {'age': 20, 'grade': 'A'},
    'Sarah': {'age': 19, 'grade': 'B'},
    'Michael': {'age': 21, 'grade': 'A+'}
}

# Accessing values in the dictionary
john_age = student_data['John']['age']
sarah_grade = student_data['Sarah']['grade']

print(f"John's age: {john_age}")
print(f"Sarah's grade: {sarah_grade}")

# Modifying values in the dictionary
student_data['Michael']['grade'] = 'A'
```

```
# Adding a new student
new_student = {'age': 18, 'grade': 'B+'}
student_data['Emily'] = new_student

# Iterating over the dictionary
for name, data in student_data.items():
    print(f"Name: {name}")
    print(f"Age: {data['age']}")
    print(f"Grade: {data['grade']}")
    print()

# Checking if a student exists
if 'John' in student_data:
    print("John is in the student data.")

# Removing a student
removed_student = student_data.pop('Sarah')

# Printing the updated dictionary
print(student_data)
```