(1) What is walrus operator in python? give one example .

Answer: Walrus operator, also known as the assignment expression, is a feature introduced in Python 3.8. It allows you to assign a value to a variable as part of an expression. The operator is represented by :=.

```
# Without walrus operator
name = input("Enter your name: ")
if len(name) > 0:
    print("Hello, " + name)
else:
    print("No name entered")

# With walrus operator
if (name := input("Enter your name: ")) != "":
    print("Hello, " + name)
else:
    print("No name entered")
```

In the above example, the walrus operator is used to assign the value of `input("Enter your name: ")` to the variable `name` and simultaneously check if the length of the name is greater than 0. This allows us to eliminate the need for a separate line of code to assign the value and then check its length.

(2) different ways to do String slicing using positive indexing and negative indexing .

Answer:

String slicing can be done using positive indexing and negative indexing in Python. Here are the different ways to perform string slicing using both types of indexing:

Positive Indexing:

1. Basic slicing: `string[start:end]` - This returns the substring starting from the `start` index and ending at the `end-1` index.
2. Slicing with step: `string[start:end:step]` - This returns the substring starting from the `start` index and ending at the `end-1` index, with a step size of `step`.

Negative Indexing:

3. Basic slicing: `string[start:end]` - This returns the substring starting from the `start` index (counted from the end of the string) and ending at the `end-1` index (also counted from the end).

4. Slicing with step: `string[start:end:step]` - This returns the substring starting from the `start` index (counted from the end of the string) and ending at the `end-1` index (also counted from the end), with a step size of `step`.

Here are some examples to illustrate the usage of positive and negative indexing for string slicing:

Positive Indexing:

```
string = "Hello, World!"

# Basic slicing
print(string[0:5])  # Output: "Hello"
print(string[7:])  # Output: "World!"

# Slicing with step
print(string[0:12:2])  # Output: "Hlo ol"
```

Negative Indexing:

```
string = "Hello, World!"

# Basic slicing
print(string[-6:-1])  # Output: "World"
print(string[:-8])  # Output: "Hello"

# Slicing with step
print(string[-1:-13:-2])  # Output: "!loW ,"
```

Note that in both positive and negative indexing, the `start` index is inclusive, while the `end` index is exclusive.

(3) / and // difference in python?

In Python, the forward slash (/) and double forward slash (//) operators are used for division, but they have different behaviors.

The forward slash (/) operator performs normal division and returns a floating-point result.

```
result = 7 / 2
print(result) # Output: 3.5
```

In this case, the result is 3.5 because 7 divided by 2 is 3.5.

On the other hand, the double forward slash (//) operator performs floor division, which returns the largest integer that is less than or equal to the division result.

```
result = 7 // 2
print(result) # Output: 3
```

In this case, the result is 3 because the largest integer less than or equal to 3.5 is 3.
To summarize, the forward slash (/) operator returns a floating-point result, while the double forward slash (//) operator returns an integer result by performing floor division

5. what are the various ways to do swap in python

Answer:

In Python, there are several ways to swap the values of two variables. Here are a few common methods:

Using a temporary variable:

```
a = 1
b = 2
temp = a
a = b
b = temp
```

Using tuple packing/unpacking:

```
a = 1
b = 2
a, b = b, a
```

Using arithmetic operations:

```
a = 1
b = 2
a = a + b
b = a - b
a = a – b
```

Using bitwise XOR operation:

```
a = 1
b = 2
a = a ^ b
b = a ^ b
a = a ^ b
```

(6). how to print current timestamp in python

Answer:

To print the current timestamp in Python, you can use the `datetime` module. Here's an

```
import datetime

current_timestamp = datetime.datetime.now()
print(current_timestamp)
```

This will output the current timestamp in the format `YYYY-MM-DD HH:MM:SS.ssssss`, where `YYYY` represents the year, `MM` represents the month, `DD` represents the day, `HH` represents the hour, `MM` represents the minute, `SS` represents the second, and `ssssss` represents the microsecond.

If you want to print the timestamp in a specific format, you can use the `strftime` method of the `datetime` object. Here's an example that prints the timestamp in the format `YYYY-MM-DD HH:MM:SS`:

```
import datetime

current_timestamp = datetime.datetime.now()
formatted_timestamp = current_timestamp.strftime("%Y-%m-%d %H:%M:%S")
print(formatted_timestamp)
```

This will output the current timestamp in the format `YYYY-MM-DD HH:MM:SS`.

(7). how to use function annotation in python ?

Answer:

Function annotations in Python are used to provide additional information about the types of arguments and the return value of a function. They are optional and do not affect the runtime behavior of the function. Here's how you can use function annotations in Python:

6. Annotating arguments: You can annotate the arguments of a function by specifying the type after the argument name, separated by a colon.

```
def greet(name: str, age: int) -> str:
    return f"Hello, {name}! You are {age} years old."
```

In this example, the `name` argument is annotated as a string (`str`), and the `age` argument is annotated as an integer (`int`).

7. Annotating the return value: You can annotate the return value of a function by specifying the type after the closing parenthesis of the argument list, separated by a right arrow (`->`).

```
def add(a: int, b: int) -> int:
    return a + b
```

In this example, the return value of the `add` function is annotated as an integer (`int`).

8. Accessing annotations: You can access the annotations of a function using the `__annotations__` attribute. It returns a dictionary where the keys are the argument names and the return value, and the values are the corresponding annotations.

```
print(greet.__annotations__)  # {'name': <class 'str'>, 'age': <class 'int'>, 'return': <class 'str'>}
```

This will print the annotations of the `greet` function.

Function annotations are primarily used for documentation purposes and can be helpful for static type checkers and IDEs that support type hinting. They do not enforce any type checking at runtime, so it's important to note that you still need to handle type validation manually if required.

(8) can function take function as function argument ? if yes, specify how?

Answer:

Yes, functions can take other functions as arguments. This is known as higher-order functions or function composition.

```
def apply_twice(func, arg):
    return func(func(arg))

def square(x):
    return x * x

result = apply_twice(square, 2)
print(result)  # Output: 16
```

In this example, the `apply_twice` function takes two arguments: `func` (which is a function) and `arg` (which is a value). It applies the `func` twice to the `arg` and returns the result.
The `square` function is defined separately and takes a single argument `x`. It squares the value of `x` and returns the result.
In the example, we pass the `square` function as an argument to the `apply_twice` function, along with the value 2. The `apply_twice` function applies the `square` function twice to the value 2, resulting in 16.

9. difference between decorator and generator.
   Answer :

Decorators and generators are both powerful features in Python, but they serve different purposes and have different syntax.
Decorators are a way to modify the behavior of a function or class without changing its source code. They allow you to wrap a function or class with additional functionality. Decorators are defined using the @ symbol followed by the decorator name, placed above the function or class definition.
Here's an example of a simple decorator that logs the execution time of a function:

```
import time

def timer_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
```

```
        print(f"Execution time: {execution_time} seconds")
        return result
    return wrapper

@timer_decorator
def my_function():
    time.sleep(2)
    print("Function executed")

my_function()
```

In this example, the `timer_decorator` function is defined, which takes a function as an argument and returns a new function (`wrapper`) that wraps the original function. The `wrapper` function measures the execution time of the original function and prints it before returning the result.

Generators, on the other hand, are a way to create iterators in Python. They allow you to generate a sequence of values on-the-fly, without storing them in memory. Generators are defined using a special kind of function called a generator function, which uses the `yield` keyword instead of `return`.

```
def fibonacci_generator():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

fib = fibonacci_generator()
for i in range(10):
    print(next(fib))
```

In this example, the `fibonacci_generator` function is defined as a generator function. It uses an infinite loop to generate the Fibonacci sequence, yielding each value one at a time. The generator is then used in a `for` loop to print the first 10 values of the sequence.

To summarize, decorators are used to modify the behavior of functions or classes, while generators are used to create iterators that generate values on-the-fly. Decorators are defined using the @ symbol above the function or class definition, while generators are defined using a generator function that uses the `yield` keyword.

10. :diff between format strings and raw strings , specify syntax and application.

    Answer:

Format strings and raw strings are two different concepts in Python.

Format Strings:

- Format strings are regular strings that contain placeholders, which are marked by curly braces {}.
- They are used to dynamically insert values into a string.
- The placeholders can be replaced with values using the `format()` method or the f-string syntax (introduced in Python 3.6).
- Format strings allow for string interpolation and formatting, such as specifying the number of decimal places, padding, alignment, etc.

```
name = "Alice"
age = 25
message = "My name is {} and I'm {} years old.".format(name, age)
print(message)  # Output: My name is Alice and I'm 25 years old.
```

Raw Strings:

- Raw strings are prefixed with the letter `r` or `R`.
- They are used to treat backslashes (\) as literal characters, rather than escape characters.
- Raw strings are often used when dealing with regular expressions, file paths, or any situation where backslashes need to be preserved.

```
path = r'C:\Users\Documents\file.txt'
print(path)  # Output: C:\Users\Documents\file.txt
```

In summary, format strings are used for string interpolation and formatting, while raw strings are used to treat backslashes as literal characters.

(11) multi-indexing in pandas

Multi-indexing in pandas refers to the ability to have multiple levels of indexing in a DataFrame or a Series. It allows you to work with data that has multiple dimensions or hierarchical structure.

To create a multi-index DataFrame, you can pass a list of arrays or tuples as the index parameter when creating a DataFrame. Each array or tuple represents a level of the index.

```
import pandas as pd

# Create a multi-index DataFrame
index = pd.MultiIndex.from_tuples([('A', 1), ('A', 2), ('B', 1), ('B', 2)])
data = {'Value': [10, 20, 30, 40]}
df = pd.DataFrame(data, index=index)
```

```
print(df)
```

Output:

```
    Value
A 1     10
  2     20
B 1     30
  2     40
```

In this example, we have a DataFrame with two levels of indexing: the first level has two categories 'A' and 'B', and the second level has two categories 1 and 2. The values in the 'Value' column correspond to each combination of the index levels.
You can access data in a multi-index DataFrame using the `.loc` accessor. For example, to select all rows where the first level of the index is 'A', you can use:

```
print(df.loc['A'])
```

Output:

```
  Value
1     10
2     20
```

You can also perform operations like grouping, aggregation, and slicing on multi-index DataFrames. The multi-indexing feature in pandas provides a powerful way to work with complex and hierarchical data structures.

(12) rolling mean in pandas

In pandas, you can calculate the rolling mean of a series or a DataFrame using the `rolling` method. The `rolling` method creates a rolling window object, and you can then apply various aggregation functions, such as mean, sum, min, max, etc., to the window.
Here's an example of calculating the rolling mean of a series:

```
import pandas as pd

# Create a sample series
data = pd.Series([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# Calculate the rolling mean with a window size of 3
rolling_mean = data.rolling(window=3).mean()
```

```
print(rolling_mean)
```

Output:

```
0    NaN
1    NaN
2    2.0
3    3.0
4    4.0
5    5.0
6    6.0
7    7.0
8    8.0
9    9.0
dtype: float64
```

In this example, the rolling mean is calculated with a window size of 3. The first two values in the resulting series are NaN because there are not enough values to calculate the mean. The rolling mean is then calculated for the remaining values by taking the mean of the current value and the two preceding values.

You can also calculate the rolling mean for multiple columns in a DataFrame by specifying the column(s) you want to calculate the rolling mean on. For example:

```
import pandas as pd

# Create a sample DataFrame
data = pd.DataFrame({'A': [1, 2, 3, 4, 5],
                     'B': [6, 7, 8, 9, 10]})

# Calculate the rolling mean for column 'A' with a window size of 2
rolling_mean = data['A'].rolling(window=2).mean()

print(rolling_mean)
```

Output:

```
0    NaN
1    1.5
2    2.5
3    3.5
4    4.5
Name: A, dtype: float64
```

In this example, the rolling mean is calculated for column 'A' with a window size of 2. The first value in the resulting series is NaN because there is not enough data to calculate the mean. The rolling mean is then calculated for the remaining values by taking the mean of the current value and the preceding value.

13.The `timedelta` function in Python is a class from the `datetime` module that represents a duration or difference between two dates or times. It allows you to perform arithmetic operations on dates and times, such as adding or subtracting a specific amount of time.

14.The time complexity of set operations in Python is generally O(1) on average, meaning that the time it takes to perform operations like adding, removing, or checking for membership in a set does not depend on the size of the set. However, in the worst case scenario, the time complexity can be O(n), where n is the number of elements in the set.

For breadth-first search (BFS), the time complexity is O(V + E), where V is the number of vertices and E is the number of edges in the graph. The space complexity of BFS is O(V), as it requires storing the visited vertices in a queue.

15.To remove and replace null values in a dataframe, you can use the `fillna()` method in pandas. Here's an example:

```
import pandas as pd

# Create a dataframe with null values
df = pd.DataFrame({'A': [1, 2, None, 4, 5],
                   'B': [None, 2, 3, None, 5]})

# Remove null values by dropping rows or columns
df.dropna()  # Drop rows with any null values
df.dropna(axis=1)  # Drop columns with any null values

# Replace null values with a specific value
df.fillna(0)  # Replace null values with 0

# Replace null values with the mean, median, or mode
df.fillna(df.mean())  # Replace null values with the mean of each column
df.fillna(df.median())  # Replace null values with the median of each column
df.fillna(df.mode().iloc[0])  # Replace null values with the mode of each column
```

Other ways to replace null values in a dataframe include using forward fill (`ffill()`) or backward fill (`bfill()`) to fill null values with the previous or next non-null value, respectively.

16. One-hot encoding is a technique used to convert categorical variables into a binary representation that can be used for machine learning algorithms. It creates new binary columns for each unique category in the original variable, where a value of 1 indicates the presence of that category and 0 indicates its absence.

Here's a simple example of one-hot encoding using the `get_dummies()` function in pandas:

```
import pandas as pd

# Create a dataframe with a categorical variable
df = pd.DataFrame({'Color': ['Red', 'Blue', 'Green', 'Red', 'Green']})

# Perform one-hot encoding
one_hot_encoded = pd.get_dummies(df['Color'])

# Concatenate the original dataframe with the one-hot encoded columns
df_encoded = pd.concat([df, one_hot_encoded], axis=1)

print(df_encoded)
```

Output:

```
  Color  Blue  Green  Red
0   Red     0      0    1
1  Blue     1      0    0
2 Green     0      1    0
3   Red     0      0    1
4 Green     0      1    0
```

In this example, the original categorical variable "Color" is one-hot encoded into three binary columns: "Blue", "Green", and "Red". Each row has a 1 in the corresponding column for the color present in that row.

17. The `set()` and `frozenset()` are both built-in data types in Python that represent an unordered collection of unique elements. The main difference between them is that `set()`

is mutable, meaning you can add, remove, or modify elements in a set, while `frozenset()` is immutable, meaning you cannot modify its elements once it is created. Here's an example to illustrate the difference:

```
my_set = set([1, 2, 3])
my_set.add(4)
print(my_set)  # Output: {1, 2, 3, 4}

my_frozenset = frozenset([1, 2, 3])
# my_frozenset.add(4)  # This line will raise an AttributeError
print(my_frozenset)  # Output: frozenset({1, 2, 3})
```

Regarding the `sort()` and `sorted()` methods in Python, they are used to sort elements in a list. However, they have some differences:

- `sort()` is a method that sorts the list in-place, meaning it modifies the original list. It does not return a new sorted list, but rather sorts the list itself.

```
my_list = [3, 1, 2]
my_list.sort()
print(my_list)  # Output: [1, 2, 3]
```

- `sorted()` is a built-in function that takes an iterable (e.g., a list) as an argument and returns a new sorted list. It does not modify the original list.

```
my_list = [3, 1, 2]
sorted_list = sorted(my_list)
print(sorted_list)  # Output: [1, 2, 3]
print(my_list)  # Output: [3, 1, 2]
```

In summary, `sort()` sorts the list in-place, while `sorted()` returns a new sorted list without modifying the original list.

18. To add values from three columns and write the result in a fourth column, you can use the following steps:

- Iterate over the rows of your dataset.
- Access the values from the three columns you want to add.
- Perform the addition operation on the values.
- Write the result in the fourth column of the corresponding row.
- Here's an example in Python using the pandas library:

```
11. import pandas as pd
```

```
    # Read the dataset into a pandas DataFrame
    df = pd.read_csv('your_dataset.csv')

    # Iterate over the rows
    for index, row in df.iterrows():
        # Access the values from the three columns
        value1 = row['column1']
        value2 = row['column2']
        value3 = row['column3']

        # Perform the addition operation
        result = value1 + value2 + value3

        # Write the result in the fourth column
        df.at[index, 'column4'] = result

    # Save the updated DataFrame to a new CSV file
    df.to_csv('updated_dataset.csv', index=False)
```

Make sure to replace `'your_dataset.csv'` with the path to your actual dataset file, and `'column1'`, `'column2'`, `'column3'`, and `'column4'` with the names of the columns you want to work with.

(19)Pickling is the process of serializing and deserializing Python objects. It allows you to convert complex objects, such as lists, dictionaries, and custom classes, into a byte stream that can be stored in a file or transferred over a network. The byte stream can later be converted back into the original object.
Python provides the `pickle` module for pickling and unpickling objects. Here's a simple example:

```
import pickle

# Create a dictionary
data = {'name': 'John', 'age': 30, 'city': 'New York'}

# Pickle the dictionary
with open('data.pickle', 'wb') as file:
    pickle.dump(data, file)

# Unpickle the dictionary
with open('data.pickle', 'rb') as file:
    loaded_data = pickle.load(file)

# Print the unpickled dictionary
print(loaded_data)
```

In this example, the `pickle.dump()` function is used to pickle the `data` dictionary and write it to a file called `'data.pickle'`. The `pickle.load()` function is then used to read the pickled data from the file and load it back into the `loaded_data` variable.

20. The best way to initialize a dictionary according to linters, such as Pylint, is to use the dictionary literal syntax. Here's an example:

```
my_dict = {}
```

Linters generally prefer this syntax because it is concise and easy to read. If you need to initialize the dictionary with some initial key-value pairs, you can do so using the literal syntax as well:

```
my_dict = {'key1': 'value1', 'key2': 'value2'}
```

Linters may also suggest using the `dict()` constructor, but the dictionary literal syntax is generally preferred for its simplicity.

(21) To set a default value for a dictionary key, you can use the `dict.get()` method. The `get()` method returns the value for a given key if it exists in the dictionary, and a default value if the key is not found. Here's an example:

```
my_dict = {'key1': 'value1', 'key2': 'value2'}

# Get the value for 'key3', with a default value of 'default_value'
value = my_dict.get('key3', 'default_value')

print(value)  # Output: 'default_value'
```

In this example, the `get()` method is used to retrieve the value for the key `'key3'`. Since `'key3'` does not exist in the dictionary, the method returns the default value `'default_value'`.
You can also set a default value for a key using the `dict.setdefault()` method. This method works similarly to `get()`, but it also sets the default value for the key if it doesn't exist in the dictionary. Here's an example:

```
my_dict = {'key1': 'value1', 'key2': 'value2'}

# Get the value for 'key3', with a default value of 'default_value'
value = my_dict.setdefault('key3', 'default_value')

print(value)  # Output: 'default_value'
print(my_dict)  # Output: {'key1': 'value1', 'key2': 'value2', 'key3': 'default_value'}
```

In this example, the `setdefault()` method is used to retrieve the value for the key `'key3'`.

(22)`set()` vs `frozenset()`:

- `set()` is a built-in Python data structure that represents an unordered collection of unique elements. It is mutable, meaning you can add or remove elements from it.
- `frozenset()` is also a built-in Python data structure that represents an immutable version of a set. Once created, you cannot modify its elements.

Docstring in function: A docstring is a string literal that appears as the first statement in a function, method, class, or module definition. It provides a way to document what the function does, what arguments it takes, and what it returns. It is used to provide documentation and help users understand how to use the function.

To write a docstring for a function, you can enclose a multi-line string in triple quotes (''' or """) immediately after the function definition. Here's an example:

```
def my_function(arg1, arg2):
    """
    This is a docstring for my_function.

    Args:
        arg1: Description of arg1.
        arg2: Description of arg2.

    Returns:
        Description of the return value.
    """
    # Function code goes here
```

(23)Combining two sets: To combine two sets, you can use the `union()` method or the `|` operator. Here's an example:

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# Using union() method
combined_set = set1.union(set2)
print(combined_set)  # Output: {1, 2, 3, 4, 5}

# Using | operator
combined_set = set1 | set2
print(combined_set)  # Output: {1, 2, 3, 4, 5}
```

(24)Head and tail in pandas: In pandas, you can use the `head()` and `tail()` methods to retrieve the first or last n rows of a DataFrame or Series.

```python
import pandas as pd

# Create a DataFrame
df = pd.DataFrame({'A': [1, 2, 3, 4, 5], 'B': ['a', 'b', 'c', 'd', 'e']})

# Get the first 3 rows
head_df = df.head(3)
print(head_df)

# Get the last 3 rows
tail_df = df.tail(3)
print(tail_df)
```

(25)Join and merge in pandas: Both `join()` and `merge()` are methods in pandas used to combine two or more DataFrames based on common columns or indices.

- `join()`: This method is used to join two DataFrames based on their indices. By default, it performs an inner join, but you can specify different types of joins using the how parameter.

```python
import pandas as pd

# Create two DataFrames
df1 = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'c']}, index=[1, 2, 3])
df2 = pd.DataFrame({'C': [4, 5, 6], 'D': ['d', 'e', 'f']}, index=[2, 3, 4])

# Join the DataFrames based on their indices
joined_df = df1.join(df2)
print(joined_df)
```

- `merge()`: This method is used to merge two DataFrames based on common columns. You can specify the columns to merge on using the on parameter.

```python
import pandas as pd

# Create two DataFrames
df1 = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'c']})
df2 = pd.DataFrame({'A': [2, 3, 4], 'C
```

(26)
```python
d=dict()
for x in enumerate(range(2)):
 d[x[0]]=x[1]
 d[x[1]+7]=x[0]
print(d)
```

Output :

{0: 0, 7: 0, 1: 1, 8: 1}