

## 1. What are Python built-in data types?

Python has several built-in data types that allow you to store and manipulate different kinds of data. Here's a comprehensive overview of Python's built-in data types:

### Numeric Types:

1. **int**: Represents integers.

```
x = 5
```

```
y = -10
```

2. **float**: Represents floating-point numbers.

```
x = 3.14
```

```
y = -2.718
```

3. **complex**: Represents complex numbers with a real and an imaginary part.

```
x = 2 + 3j
```

```
y = -5j
```

### Sequence Types:

4. **str**: Represents strings, which are sequences of Unicode characters.

```
s = "Hello, World!"
```

5. **list**: Represents ordered, mutable sequences of objects.

```
list = [1, 2, 3, "a", "b", "c"]
```

6. **tuple**: Represents ordered, immutable sequences of objects.

```
tup = (1, 2, 3, "a", "b", "c")
```

7. **range**: Represents an immutable sequence of numbers, often used for looping a specific number of times in for loops.

```
r = range(10)
```

### Mapping Types:

8. **dict**: Represents key-value pairs (unordered, changeable, and indexed).

```
d = {"name": "Alice", "age": 25}
```

### Set Types:

9. **set**: Represents unordered collections of unique elements.

```
s = {1, 2, 3, 4, 5}
```

10. **frozenset**: Represents immutable sets.

```
fs = frozenset([1, 2, 3, 4, 5])
```

### Boolean Type:

11. **bool**: Represents Boolean values (True or False).

```
b1 = True
```

```
b2 = False
```

### Binary Types:

12. **bytes**: Represents immutable sequences of bytes.

```
b = b"Hello"
```

13. **bytearray**: Represents mutable sequences of bytes.

```
ba = bytearray(b"Hello")
```

14. **memoryview**: Provides a way to access the internal data of an object that supports the buffer protocol without copying.

```
mv = memoryview(bytes(5))
```

### None Type:

15. **NoneType**: Represents the absence of a value or a null value.

```
n = None
```

Each of these data types is designed for specific use cases and has its own methods and properties that make them versatile for different programming tasks in Python.

## 2. What are lists and tuples?

Lists and tuples are both sequence data types in Python that allow you to store collections of items. They share some similarities but also have key differences.

### Lists:

- **Mutable**: You can change, add, or remove items after the list is created.
- **Ordered**: The items have a defined order, and that order will not change unless explicitly changed.
- **Dynamic Size**: Lists can grow and shrink as needed.
- **Flexible**: They can hold items of different data types, including other lists.

### Creating a List:

```
# Creating a list
```

```
my_list = [1, 2, 3, "a", "b", "c"]
```

### Accessing Elements:

```
# Accessing elements
```

```
print(my_list[0]) # Output: 1
```

```
print(my_list[-1]) # Output: "c"
```

### **Modifying a List:**

```
# Modifying elements
```

```
my_list[1] = "changed"
```

```
print(my_list) # Output: [1, "changed", 3, "a", "b", "c"]
```

```
# Adding elements
```

```
my_list.append("new")
```

```
print(my_list) # Output: [1, "changed", 3, "a", "b", "c", "new"]
```

```
# Removing elements
```

```
my_list.remove("a")
```

```
print(my_list) # Output: [1, "changed", 3, "b", "c", "new"]
```

### **Common Methods:**

- `append()`: Adds an element to the end.
- `extend()`: Adds all elements of an iterable to the end.
- `insert()`: Inserts an element at a specific position.
- `remove()`: Removes the first occurrence of a specific element.
- `pop()`: Removes and returns the element at a specific position.
- `sort()`: Sorts the list in place.
- `reverse()`: Reverses the elements of the list in place.

### **Tuples:**

- **Immutable**: Once created, you cannot change, add, or remove items from a tuple.
- **Ordered**: The items have a defined order, and that order will not change.
- **Fixed Size**: The size of a tuple is fixed after creation.
- **Flexible**: They can hold items of different data types, including other tuples.

### **Creating a Tuple:**

```
# Creating a tuple
```

```
my_tuple = (1, 2, 3, "a", "b", "c")
```

### **Accessing Elements:**

```
# Accessing elements
```

```
print(my_tuple[0]) # Output: 1
```

```
print(my_tuple[-1]) # Output: "c"
```

### Attempting to Modify a Tuple:

# Trying to modify elements (this will raise an error)

# my\_tuple[1] = "changed" # TypeError: 'tuple' object does not support item assignment

### Common Uses:

- Tuples are often used for fixed collections of items.
- They can be used as keys in dictionaries (because they are immutable).
- They are used to group data and return multiple values from a function.

### Summary:

- **Lists:** Mutable, dynamic-sized, ordered collections that are highly versatile and widely used in scenarios where modification of the sequence is required.
- **Tuples:** Immutable, fixed-sized, ordered collections used primarily when a constant set of values is needed and where immutability is a requirement, such as keys in dictionaries or returning multiple values from functions.

### 3.What is a dictionary in python?

In Python, a dictionary is a mutable, unordered collection of key-value pairs. Each key is unique within the dictionary, and keys are used to access their corresponding values. Dictionaries are versatile and efficient for storing and retrieving data where relationships between elements are essential.

### Key Features of Dictionaries:

- **Mutable:** You can change, add, or remove items after the dictionary is created.
- **Unordered:** The items do not have a defined order. From Python 3.7 onwards, dictionaries maintain the insertion order.
- **Keys:** Must be unique and immutable (e.g., strings, numbers, or tuples).
- **Values:** Can be of any data type and can be duplicated.

### Creating a Dictionary:

# Creating a dictionary

```
my_dict = {  
    "name": "Alice",  
    "age": 25,  
    "city": "New York"  
}
```

### Accessing Values:

# Accessing values using keys

```
print(my_dict["name"]) # Output: Alice
```

```
print(my_dict["age"]) # Output: 25
```

### **Modifying a Dictionary:**

# Adding a new key-value pair

```
my_dict["job"] = "Engineer"
```

```
print(my_dict) # Output: {'name': 'Alice', 'age': 25, 'city': 'New York', 'job': 'Engineer'}
```

# Modifying an existing value

```
my_dict["age"] = 26
```

```
print(my_dict) # Output: {'name': 'Alice', 'age': 26, 'city': 'New York', 'job': 'Engineer'}
```

# Removing a key-value pair

```
del my_dict["city"]
```

```
print(my_dict) # Output: {'name': 'Alice', 'age': 26, 'job': 'Engineer'}
```

### **Dictionary Methods:**

- `dict.get(key, default)`: Returns the value for the specified key if key is in the dictionary, otherwise returns default.

```
print(my_dict.get("name")) # Output: Alice
```

```
print(my_dict.get("city", "N/A")) # Output: N/A
```

- `dict.keys()`: Returns a view object that displays a list of all the keys.

```
print(my_dict.keys()) # Output: dict_keys(['name', 'age', 'job'])
```

- `dict.values()`: Returns a view object that displays a list of all the values.

```
print(my_dict.values()) # Output: dict_values(['Alice', 26, 'Engineer'])
```

- `dict.items()`: Returns a view object that displays a list of dictionary's key-value tuple pairs.

```
print(my_dict.items()) # Output: dict_items([('name', 'Alice'), ('age', 26), ('job', 'Engineer')])
```

- `dict.update([other])`: Updates the dictionary with the key-value pairs from another dictionary or from an iterable of key-value pairs.

```
my_dict.update({"city": "Boston", "age": 27})
```

```
print(my_dict) # Output: {'name': 'Alice', 'age': 27, 'job': 'Engineer', 'city': 'Boston'}
```

- `dict.pop(key, default)`: Removes the specified key and returns the corresponding value. If the key is not found, returns default.

```
job = my_dict.pop("job", "N/A")
```

```
print(job) # Output: Engineer
```

```
print(my_dict) # Output: {'name': 'Alice', 'age': 27, 'city': 'Boston'}
```

### Example Use Case:

Dictionaries are particularly useful when you need to associate pieces of information. For example, you could use a dictionary to store user profiles:

python

Copy code

```
user_profiles = {  
    "user1": {"name": "Alice", "age": 25, "city": "New York"},  
    "user2": {"name": "Bob", "age": 30, "city": "San Francisco"}  
}
```

# Accessing a user's profile

```
print(user_profiles["user1"]["name"]) # Output: Alice
```

Dictionaries provide a powerful and flexible way to manage data with key-value relationships, making them an essential part of Python programming.

4.Explain list comprehensions with an example?

ChatGPT

List comprehensions in Python provide a concise way to create lists. They consist of brackets containing an expression followed by a for clause, and potentially additional for or if clauses. This approach reduces the need for traditional for loop syntax, making the code cleaner and often more readable.

### Basic Syntax:

[expression for item in iterable if condition]

- **expression:** The value or operation applied to each item.
- **item:** The variable representing the current item from the iterable.
- **iterable:** A collection of items (e.g., list, range, string) to iterate over.
- **condition (optional):** A filter that applies to each item.

### Example 1: Simple List Comprehension

Creating a list of squares of numbers from 0 to 9:

python

Copy code

```
squares = [x**2 for x in range(10)]  
  
print(squares) # Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Here, `x**2` is the expression, `x` is the item, and `range(10)` is the iterable.

### Example 2: List Comprehension with Condition

Creating a list of even numbers from 0 to 9:

```
evens = [x for x in range(10) if x % 2 == 0]
```

```
print(evens) # Output: [0, 2, 4, 6, 8]
```

- The condition `if x % 2 == 0` filters out odd numbers.

### Example 3: Nested List Comprehension

Creating a list of tuples representing coordinates (x, y) where both x and y range from 0 to 2:

```
coordinates = [(x, y) for x in range(3) for y in range(3)]
```

```
print(coordinates) # Output: [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

- Here, two for clauses iterate over both x and y.

### Example 4: Complex Expression and Condition

Creating a list of lowercase vowels from a string, but only if they are vowels:

```
sentence = "Hello World"
```

```
vowels = [char.lower() for char in sentence if char.lower() in 'aeiou']
```

```
print(vowels) # Output: ['e', 'o', 'o']
```

- The expression `char.lower()` ensures all characters are lowercase.
- The condition `if char.lower() in 'aeiou'` filters only vowels.

### Example 5: Using a Function in List Comprehension

Applying a function to each element in a list:

```
def square(x):
```

```
    return x * x
```

```
numbers = [1, 2, 3, 4, 5]
```

```
squared_numbers = [square(x) for x in numbers]
```

```
print(squared_numbers) # Output: [1, 4, 9, 16, 25]
```

- The `square(x)` function is applied to each element in `numbers`.

List comprehensions provide a powerful and efficient way to generate and manipulate lists in Python, making your code more concise and readable.

5.What are some common data cleaning techniques?

Data cleaning is a crucial step in data preparation, ensuring that the data you work with is accurate, complete, and usable. Here are some common data cleaning techniques:

#### 1. Handling Missing Data:

- **Imputation:** Replace missing values with a specific value, such as the mean, median, or mode of the column, or use advanced methods like k-nearest neighbors (KNN) or regression imputation.

# Using pandas to fill missing values with the mean

```
df['column'].fillna(df['column'].mean(), inplace=True)
```

- **Removal:** Delete rows or columns with missing values if they are not significant or if their absence does not affect the analysis.

# Removing rows with missing values

```
df.dropna(inplace=True)
```

## 2. Removing Duplicates:

- **Identify and Remove Duplicates:** Use methods to find and remove duplicate rows from the dataset.

# Removing duplicate rows

```
df.drop_duplicates(inplace=True)
```

## 3. Handling Outliers:

- **Detection:** Identify outliers using statistical methods (e.g., Z-score, IQR) or visual methods (e.g., box plots).

# Using IQR to detect outliers

```
Q1 = df['column'].quantile(0.25)
```

```
Q3 = df['column'].quantile(0.75)
```

```
IQR = Q3 - Q1
```

```
outliers = df[(df['column'] < (Q1 - 1.5 * IQR)) | (df['column'] > (Q3 + 1.5 * IQR))]
```

- **Treatment:** Remove or transform outliers based on their impact on the analysis.

# Removing outliers

```
df = df[~df['column'].isin(outliers['column'])]
```

## 4. Data Type Conversion:

- **Convert Data Types:** Ensure that each column has the appropriate data type (e.g., convert strings to datetime or numeric types).

# Converting a column to datetime

```
df['date_column'] = pd.to_datetime(df['date_column'])
```

## 5. Standardization and Normalization:

- **Standardization:** Scale features so they have a mean of 0 and a standard deviation of 1.

# Standardizing data



```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

df['scaled_column'] = scaler.fit_transform(df[['column']])
```

- **Normalization:** Scale features to a specific range, often between 0 and 1.

```
# Normalizing data
```

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

df['normalized_column'] = scaler.fit_transform(df[['column']])
```

## 6. Handling Categorical Data:

- **Encoding:** Convert categorical values into numerical formats (e.g., one-hot encoding, label encoding).

```
# One-hot encoding
```

```
df = pd.get_dummies(df, columns=['categorical_column'])
```

- **Binning:** Convert continuous variables into categorical bins.

```
# Binning continuous data
```

```
df['binned_column'] = pd.cut(df['column'], bins=[0, 10, 20, 30], labels=['Low', 'Medium', 'High'])
```

## 7. Correcting Inconsistencies:

- **Standardize Text:** Correct inconsistencies in text data (e.g., different spellings, case sensitivity).

```
# Standardizing text
```

```
df['text_column'] = df['text_column'].str.lower().str.strip()
```

- **Consistent Formatting:** Ensure that data formats are consistent (e.g., date formats, phone numbers).

## 8. Removing Irrelevant Data:

- **Feature Selection:** Remove columns that are not relevant to the analysis or model.

```
# Dropping irrelevant columns
```

```
df.drop(columns=['irrelevant_column'], inplace=True)
```

## 9. Handling Data Entry Errors:

- **Correct Errors:** Fix inaccuracies or incorrect entries based on domain knowledge or external data sources.

```
# Correcting specific errors
```

```
df.loc[df['column'] == 'wrong_value', 'column'] = 'correct_value'
```

## 10. Data Aggregation:

- **Aggregation:** Combine data from different sources or summarize data at different levels.

# Aggregating data

```
df_agg = df.groupby('category').agg({'value': 'sum'})
```

Effective data cleaning involves a combination of these techniques, tailored to the specific characteristics and needs of your dataset. Proper data cleaning ensures that the analysis, modeling, and insights derived from the data are accurate and reliable.

6. How do you handle missing values in a dataset ?

Handling missing values in a dataset is a critical part of data cleaning. The method you choose depends on the nature of the data, the amount of missing data, and the intended analysis. Here are some common strategies for handling missing values:

### 1. Identifying Missing Values

Before handling missing values, you need to identify them. In Python, using pandas, you can find missing values with:

```
import pandas as pd
```

```
# Checking for missing values
```

```
missing_values = df.isnull().sum()
```

```
print(missing_values)
```

### 2. Removing Missing Values

#### a. Remove Rows with Missing Values:

If missing data is minimal and its removal won't affect the analysis, you can drop rows with missing values:

```
# Drop rows with any missing values
```

```
df.dropna(inplace=True)
```

```
# Drop rows with missing values in specific columns
```

```
df.dropna(subset=['column_name'], inplace=True)
```

#### b. Remove Columns with Missing Values:

If an entire column has a lot of missing values and is not useful, consider removing the column:

```
# Drop columns with any missing values
```

```
df.dropna(axis=1, inplace=True)
```

```
# Drop columns with all missing values
```

```
df.dropna(axis=1, how='all', inplace=True)
```

### 3. Imputing Missing Values

#### **a. Mean/Median/Mode Imputation:**

For numerical data, replace missing values with the mean, median, or mode of the column:

```
# Replace missing values with the mean
df['column_name'].fillna(df['column_name'].mean(), inplace=True)

# Replace missing values with the median
df['column_name'].fillna(df['column_name'].median(), inplace=True)

# Replace missing values with the mode
df['column_name'].fillna(df['column_name'].mode()[0], inplace=True)
```

#### **b. Imputation Using Algorithms:**

Use machine learning algorithms to predict and fill missing values based on other data in the dataset:

```
from sklearn.impute import SimpleImputer

# Impute missing values using the mean strategy
imputer = SimpleImputer(strategy='mean')

df[['column_name']] = imputer.fit_transform(df[['column_name']])
```

#### **c. Forward/Backward Fill:**

For time series data or ordered data, you might use forward fill (propagate the last valid observation) or backward fill (propagate the next valid observation):

```
# Forward fill
df.fillna(method='ffill', inplace=True)

# Backward fill
df.fillna(method='bfill', inplace=True)
```

#### **d. Interpolation:**

Use interpolation techniques to estimate missing values:

```
# Linear interpolation
df.interpolate(method='linear', inplace=True)
```

### **4. Using a Default Value**

In some cases, filling missing values with a specific default value (e.g., 0, "Unknown") is appropriate:

```
# Fill missing values with a default value
df['column_name'].fillna('Unknown', inplace=True)
```

### **5. Predictive Modeling**

Use advanced techniques such as K-Nearest Neighbors (KNN) or regression to predict and fill missing values based on patterns in the data:

```
from sklearn.impute import KNNImputer

# KNN imputation

knn_imputer = KNNImputer(n_neighbors=5)

df[['column_name']] = knn_imputer.fit_transform(df[['column_name']])
```

## 6. Domain-Specific Methods

Sometimes domain knowledge can guide the best approach for handling missing values, such as using industry-specific rules or external data sources.

### Summary

- **Removing Data:** Use when missing data is minimal or irrelevant.
- **Imputation:** Use statistical measures, machine learning, or algorithms when missing data is significant or data integrity is crucial.
- **Default Values:** Use when a placeholder value is suitable for the context.
- **Predictive Models:** Use for complex datasets where other imputation methods are insufficient.