Type Identification

In Python, type identification refers to the ability to determine the type of an object or variable at runtime. Python provides several built-in functions and methods for type identification, allowing you to check the type of an object dynamically. This is particularly useful in situations where you want to perform different actions based on the type of an object.

1. `type()` Function:

The `type()` function returns the type of an object. It takes a single argument, and you can use it to check the type of variables or instances.

Example:

```python
# Checking the type of variables
x = 5
y = "Hello"
z = [1, 2, 3]

print(type(x))  # Output: <class 'int'>
print(type(y))  # Output: <class 'str'>
print(type(z))  # Output: <class 'list'>

# Checking the type of instances
class MyClass:
    pass

obj = MyClass()
print(type(obj))  # Output: <class '__main__.MyClass'>
```

2. `isinstance()` Function:

The `isinstance()` function checks if an object is an instance of a specified class or a tuple of classes. It returns `True` if the object is an instance of any of the specified classes.

Example:

```python
# Checking if objects are instances of specific classes
x = 5
y = "Hello"
z = [1, 2, 3]

print(isinstance(x, int))       # Output: True
print(isinstance(y, str))       # Output: True
print(isinstance(z, (list, str)))  # Output: True (because it's a list)
```

```
# Checking if an object is an instance of a custom class
class MyClass:
    pass

obj = MyClass()
print(isinstance(obj, MyClass))  # Output: True
```

3. `__class__` Attribute:

Every object in Python has a `__class__` attribute that refers to its class. This attribute can be used to access the class of an object.

Example:

```python
# Using the __class__ attribute to check the class of an object
x = 5
y = "Hello"
z = [1, 2, 3]

print(x.__class__)  # Output: <class 'int'>
print(y.__class__)  # Output: <class 'str'>
print(z.__class__)  # Output: <class 'list'>

# Checking the class of instances
class MyClass:
    pass

obj = MyClass()
print(obj.__class__)  # Output: <class '__main__.MyClass'>
```

4. `type()` with `__class__` attribute:

Combining the `type()` function and the `__class__` attribute can be useful for type identification, especially when working with objects dynamically.

Example:

```python
# Checking the type of an object using type() and __class__
x = 5
y = "Hello"
z = [1, 2, 3]

print(type(x) is int)        # Output: True
print(type(y) is str)        # Output: True
print(type(z) is list)       # Output: True
```

```
# Checking the type of instances
class MyClass:
    pass

obj = MyClass()
print(type(obj) is MyClass)    # Output: True
```

Type identification is commonly used in scenarios where you need to handle different types of objects differently, such as in polymorphism or when processing user input in a flexible manner. It provides a way to dynamically adapt your code based on the types of objects encountered during runtime.