Type Conversion in Python

In this article, we will explore **type conversion** in Python. Often, you may have data in one type and need to convert it to another type. For example, you might have data in a **tuple**, which is *immutable*, and want to convert it into a **list** so that you can make changes. Python provides mechanisms to perform these conversions seamlessly.

Types of Type Conversion

There are two main ways to perform type conversion in Python:

- **Implicit Type Conversion**: This happens automatically without the programmer writing any specific code for it.
- **Explicit Type Conversion**: This requires the programmer to use specific functions or syntax to convert one type to another.

Implicit Type Conversion

In Python, **implicit type conversion** occurs when you combine different data types in operations, and Python automatically converts one type to another to prevent data loss.

For example, consider the following code:

```
a = 10
b = 1.5
c = a + b
print(c)  # Output: 11.5
d = True
e = a + d
print(e)  # Output: 11
```

Here, *a* is an **integer** and *b* is a **float**. When you add them, Python automatically converts the integer to a float, resulting in a float value for *c*. Similarly, *d* is a **Boolean**, and when added to an integer, it is converted to an integer (where `True` becomes `1`), resulting in *e*.

Explicit Type Conversion

Explicit type conversion requires the use of functions to convert one data type to another. Python provides several built-in functions for this purpose, such as `int()`, `float()`, `str()`, `list()`, `tuple()`, and `set()`.

Consider the following example:

```
s = "135"
i = 10 + int(s)
f = float(s)
print(i)  # Output: 145
print(f)  # Output: 135.0
```

In this example, *s* is a **string** that holds an integer value. Using `int(s)`, we convert the string to an integer and add it to *10*, resulting in *145*. Similarly, using `float(s)` converts the string to a float, resulting in *135.0*. If you try to add a string directly to an integer without conversion, Python will raise an error.

Converting Between Different Containers

You can also convert between different container types such as **list**, **tuple**, and **set**. Here's how:

```
s = "geeks"
print(list(s))    # Output: ['g', 'e', 'e', 'k', 's']
print(tuple(s))   # Output: ('g', 'e', 'e', 'k', 's')
print(set(s))     # Output: {'g', 'e', 'k', 's'}
```

Converting a string to a list or tuple breaks it down into individual characters. However, converting to a set removes any duplicate characters and does not preserve the order since sets are unordered collections.

String Representation of Containers

You can convert containers like lists, tuples, and sets into their string representations using the `str()` function.

```
l = ['a', 'b', 'c']
print(str(l))        # Output: ['a', 'b', 'c']
a = 10
b = 11
print(str(a) + str(b))   # Output: 1011
c = 12.5
print(str(c))        # Output: 12.5
```

When you convert a list to a string, you get its string representation. Concatenating two strings using the `+` operator combines them. For example, converting integers *a* and *b* to strings and adding them results in the string "1011" instead of the numerical sum *21*.

Converting Containers to Lists

Converting different containers to lists can be useful for creating ordered sequences from unordered collections like sets.

```
t = (10, 20, 30)
print(list(t))    # Output: [10, 20, 30]
s = {10, 20, 30}
print(list(s))    # Output: [10, 20, 30]
```

Here, a **tuple** and a **set** are both converted to lists. Note that while tuples maintain order, sets do not. However, once converted to a list, the order is preserved in the list.

Binary, Octal, and Hexadecimal Conversions

Python allows you to convert integers to their binary, octal, and hexadecimal representations using the `bin()`, `oct()`, and `hex()` functions, respectively.

```
a = 20
print(bin(a))   # Output: 0b10100
print(hex(a))   # Output: 0x14
print(oct(a))   # Output: 0o24
```

The prefixes `0b`, `0x`, and `0o` indicate binary, hexadecimal, and octal numbers, respectively. These prefixes help distinguish these representations from standard decimal integers.

Converting Back to Decimal

To convert binary, octal, or hexadecimal strings back to decimal integers, you can use the `int()` function with the appropriate base.

```
a = "1001"
print(int(a, 2))    # Output: 9
b = "12"
print(int(b, 8))    # Output: 10
c = "A1"
print(int(c, 16))   # Output: 161
```

Here, the `int()` function converts strings representing binary, octal, and hexadecimal numbers back to their decimal equivalents by specifying the base as the second argument.