# 1
# Variables and Types

In Python, everything is an object, and every object has a type. Types define the nature of data stored in variables, dictating what operations can be performed on them and how they behave. Unlike many other programming languages, Python is dynamically typed, meaning you don't need to declare the type of a variable explicitly. The type is determined automatically at runtime based on the value assigned to it.

Python's type system is also strong, meaning you cannot perform operations on incompatible types without explicit conversion. For example, adding a string and an integer will result in an error unless you explicitly convert one type to the other.

Python offers a rich variety of built-in types, which are the building blocks for handling data. These range from primitive types like numbers and strings to complex types like lists, dictionaries, and custom user-defined classes. Understanding types is essential because it allows developers to write clear, robust, and efficient code.

In modern Python, type hints have been introduced to help developers specify the types of variables, function arguments, and return values explicitly. Although these annotations are optional and Python remains dynamically typed, they enhance code readability and enable tools like linters and type checkers to catch potential issues before runtime.

This chapter will explore:

- Variables
- Types
- Dynamically and strong typing
- Primitive types
- Variable reassignment
- type()
- id()
- isinstance()
- Type casting

As we delve deeper into Python's type system, it is essential to start with the basics: variables. Variables act as named references to data stored in memory, allowing us to manipulate and use values effectively within a program. In the next section, we will explore what variables are, how they are assigned and reassigned, and how Python handles their identity and type dynamically. Understanding these concepts will provide a strong foundation for working with Python's flexible yet powerful type system.

## What is a Variable

A variable in programming is a named storage location in memory that holds a value. It serves as a reference to data, allowing you to perform operations on that data throughout your program. Variables are essential components of any programming language, characterized primarily by four key properties: name, value, type, and memory location.

### Has a name

The name of a variable is the identifier used to reference it in the code. In Python, variable names must follow certain rules:

- They must begin with a letter (a-z, A-Z) or an underscore (_).
- They can only contain alphanumeric characters and underscores.
- They are case-sensitive, so myVar and myvar are different variables.

For example:

```
x = 10

name = "Alice"
```

Here, x and name are the variable names.

### Has a value

The value is the data stored in the variable. This is the actual content that the variable represents, such as a number, string, or object. In Python, a variable’s value can change over time because it is dynamically typed. For example:

```
x = 10       # x has a value of 10

x = "hello"  # x now has a value of "hello"
```

### Has a type

The type defines the kind of value a variable holds. It determines what operations can be performed on the variable. Common types in Python include:

- **Numbers**: int, float, complex
- **Text**: str
- **Boolean**: bool
- **Collections**: list, tuple, dict, set, frozenset, range, bytes, bytearray, memoryview

You can check a variable's type using the type() function:

```
x = 10

print(type(x))  # <class 'int'>


y = "hello"
```

```
print(type(y))  # <class 'str'>
```

### Has a location

Every variable is stored at a specific memory location. This location is where the value of the variable is physically stored. In Python, you can use the id() function to get the memory address of a variable:

```
x = 10

print(id(x))  # Outputs the memory address of x
```

The location is managed by Python's memory management system, and as a user, you don't typically interact with it directly.

Understanding these four properties of variables helps you write efficient and clear Python code while leveraging the language's dynamic and flexible nature. Variables provide the foundation for managing data and interacting with it throughout your programs.

### Key Takeaways

- A variable is a named reference to a value in memory, enabling data manipulation in a program.
- It has four key properties: name (identifier), value (stored data), type (data classification), and memory location (physical storage).
- Python variables are dynamically typed, allowing values to change without explicit type declarations.
- Memory management is automatic, with id() providing access to a variable's memory address.

Understanding variables helps in writing efficient, flexible, and maintainable Python code. In the next section, we will explore how types function, their role in defining valid operations.

## What is a Type

In programming, a type serves as a fundamental classification system for data, defining both the set of values a variable can hold and the operations that can be performed on those values. By categorizing data into distinct types, programming languages enforce rules that prevent invalid operations, enhance code clarity, and enable compiler optimizations. Understanding types is essential for writing reliable and efficient programs, as they dictate how data is stored, manipulated, and interpreted within a system.

### Types as a Set of Values

A type in programming defines a specific set of values that a variable can hold. For example, an integer type includes values like -1, 0, or 42, while a string type consists of sequences of characters, such as "hello" or "world". Each type establishes clear boundaries, ensuring that data adheres to expected formats and behaviors.

### Types as a Set of Allowed Operations

Beyond defining values, a type also determines the operations that can be performed on those values. For instance, integers support arithmetic operations like addition, subtraction, and multiplication, whereas strings allow concatenation, slicing, and searching for substrings. By restricting operations to those that are meaningful for a given type, programming languages help maintain correctness and efficiency in code execution.

**Key Takeaways**

- A type defines the set of values a variable can hold, ensuring data consistency and structure.
- Types establish boundaries, preventing invalid data from being assigned to variables.
- Operations are determined by type, meaning only appropriate actions can be performed (e.g., arithmetic on numbers, concatenation on strings).
- Types help maintain correctness, ensuring that data is used in meaningful ways and reducing runtime errors.
- Programming languages enforce type rules, improving efficiency and allowing for compiler optimizations.

## Primitive Types

In Python, primitive types are the basic building blocks for representing simple values like numbers, text, and logical states. The four primitive types are,  integer, floating-point number, boolean, and string.

**Int**: An integer is a whole number without a fractional part, such as 42 or -7. For instance, x = 42 defines x as an integer. You can perform arithmetic operations like addition (x + 1 results in 43) or multiplication (x * 2 gives 84).

**Float**: A floating-point number represents real numbers, including those with decimal points, such as 3.14 or -0.001. For example, y = 3.14 sets y as a float. Operations like division are often more precise with floats, as 7 / 2 results in 3.5.

**Bool**: Booleans represent logical states and can only take two values: True or False. They are often used in conditions and comparisons. For instance, is_valid = (5 > 3) assigns True to is_valid because the comparison 5 > 3 is true.

**String**: Strings are sequences of characters used to represent text, such as "hello" or "123". You can concatenate strings, like "hello" + " world", which results in "hello world", or extract parts of a string using slicing, as in "hello"[1:4], which gives "ell". Strictly speaking, str is a sequence type in Python, but it behaves similarly to a primitive type because it is immutable and widely used as a fundamental data type.

These types form the foundation of Python programming, allowing you to store, manipulate, and reason about different kinds of data in your code.

## Isinstance()

The `isinstance()` function in Python is used to check if an object belongs to a specific data type or a tuple of types. It returns True if the object matches the specified type and False otherwise.

The syntax for `isinstance()` is straightforward. It takes an object and a type as arguments, such as `isinstance(object, type)`, or it can check against multiple types at once using `isinstance(object, (type1, type2, ...))`.

For example, if you define `num = 10`, calling `isinstance(num, int)` will return True because num is an integer. If you have a variable value = 3.14, `calling isinstance(value, (int, float))` will also return True since value is a float, which is included in the checked types.

Example, checking a single type:

```
num = 10

print(isinstance(num, int))
```

Example, checking multiple types:

```
value = 3.14

print(isinstance(value, (int, float)))
```

Using `isinstance()` is helpful for ensuring type safety when writing functions, as it allows dynamic type checking. It also supports checking against multiple types at once, making it a flexible tool for handling different kinds of input.

## Type casting

Type casting in Python refers to the process of converting a variable from one data type to another. Python provides built-in functions for explicit type conversion, allowing developers to transform data types as needed. Since Python is dynamically typed, variables do not have fixed types, making type casting especially useful when dealing with user input, mathematical operations, or string manipulations.

**Explicit Type Casting**

Python provides several functions for type conversion:

- int() – Converts a value to an integer.
- float() – Converts a value to a floating-point number.
- str() – Converts a value to a string.
- bool() – Converts a value to a boolean (True or False).

For example, consider converting an integer to a float and then back to an integer:

```
num = 42

num_float = float(num)  # Converts 42 to 42.0

num_int = int(num_float)  # Converts 42.0 back to 42


print(num_float)  # 42.0

print(num_int)  # 42
```

**String to Number Conversion**

Strings containing numeric values can be converted into integers or floats. However, attempting to convert a non-numeric string will raise a ValueError.

```
num_str = "100"

num_int = int(num_str)   # Converts "100" to 100

num_float = float(num_str)   # Converts "100" to 100.0


print(num_int)   # 100

print(num_float)   # 100.0
```

**Number to String Conversion**

Converting numbers to strings is useful when concatenating text with numbers.

```
age = 30

message = "I am " + str(age) + " years old."

print(message)   # "I am 30 years old."
```

**Boolean Casting**

Python treats some values as False when converted to bool, such as 0, None, "", and empty collections ([], {}, set()). All other values return True.

```
print(bool(0))   # False

print(bool(42))   # True

print(bool(""))   # False

print(bool("Python"))   # True
```

**Automatic Type Conversion (Implicit Casting)**

Python sometimes performs automatic type conversion (implicit casting) in expressions to avoid data loss.

```
num = 10    # Integer

result = num + 5.5   # Python automatically converts num to float

print(result)   # 15.5
```

Here, Python converts 10 (int) to 10.0 (float) to match the type of 5.5, ensuring accurate calculations.

Type casting is an essential part of Python programming, allowing seamless data manipulation between integers, floats, strings, and booleans. While implicit casting simplifies calculations, explicit type conversion using functions like int(), float(), str(), and bool() gives developers greater control over data representation and prevents unexpected errors.

## Key Takeaways

- Type casting converts a variable from one data type to another, enabling flexibility in data manipulation.
- Explicit type casting is done using functions like int(), float(), str(), and bool() to manually convert data types.
- Strings containing numbers can be converted to integers or floats, but non-numeric strings will raise a ValueError.
- Numbers can be converted to strings for concatenation in text-based operations.
- Boolean casting follows specific rules, treating 0, None, "", and empty collections as False, while all other values are True.
- Python supports implicit type conversion (e.g., integers are automatically converted to floats in mixed-type operations).

Understanding type casting prevents errors, especially when handling user input, mathematical operations, or data processing. In the next we will address reassigning and it's potential pitfalls.

## Reassigning

Reassigning a variable in Python means assigning a new value to a variable that already holds a value. This replaces the old value with the new one. Python allows reassignment because it is dynamically typed, meaning the type of a variable can change depending on the value assigned to it.

For example:

```
x = 10  # Initially, x is an integer

x = "hello"  # Now, x is reassigned to a string

print(x)  # Outputs: hello
```

In this case, the type of x changes from an integer to a string due to reassignment. While this flexibility is convenient, it can also introduce risks.

### Why Reassignment Can Be Dangerous

Reassignment leads to the loss of the original value. If the previous value is required later, it must be stored in another variable beforehand. Consider this example:

```
data = [1, 2, 3]

data = "Overwritten"

print(data)  # Outputs: Overwritten
```

Here, the list [1, 2, 3] is lost because the variable data was reassigned to a string. This may cause problems if the list was needed later in the program.

Reassigning a variable to a different type can also cause confusion and runtime errors. For instance:

```
value = [1, 2, 3]

value = len(value)  # Now, value is an integer
```

```
print(value[0])  # Raises an error: 'int' object is not
subscriptable
```

In this example, value was reassigned from a list to an integer, leading to a runtime error when trying to access it as a list.

### When to Use Reassignment

Reassignment is appropriate when the variable logically represents evolving data. For example, counters in loops are a common and safe use case:

```
counter = 0

for i in range(5):

    counter += 1  # Reassigning counter

print(counter)  # Outputs: 5
```

Similarly, accumulators in algorithms often require reassignment:

```
total = 0

for num in [1, 2, 3, 4]:

    total += num  # Reassigning total to accumulate the sum

print(total)  # Outputs: 10
```

Reassignment is also useful when optimizing memory usage by reusing a variable for intermediate results:

```
x = 5

x = x * 2  # Reuse x to store the doubled value

print(x)  # Outputs: 10
```

### Mitigating Risks

To reduce the risks of reassignment, use descriptive variable names that reflect the current value's purpose, and avoid changing types unless absolutely necessary. For instance, instead of reusing data, create new variables:

```
data = [1, 2, 3]

length = len(data)  # Use a new variable

print(length)  # Outputs: 3
```

By avoiding unnecessary reassignment, you make the code more readable and easier to debug, ensuring clarity about what each variable represents at any point in the program.

## Key Takeaways

- Reassignment replaces a variable’s old value with a new one, allowing dynamic changes in data.

- Python’s dynamic typing enables reassignment across different types, but this can introduce confusion and runtime errors.
- Reassignment causes loss of the previous value, so critical data should be stored elsewhere if needed later.
- Changing a variable’s type through reassignment can lead to errors, such as trying to access an integer as a list.
- Reassignment is useful in evolving data scenarios, such as counters in loops or accumulators in calculations.
- To reduce risks, use descriptive variable names and avoid reassigning different types unless necessary.
- Creating new variables instead of overwriting existing ones improves readability and prevents unintended side effects.

## Summary

In Python, everything is an object, and every object has a type that defines its behavior and the operations it supports. Python is dynamically typed, so the type of a variable is determined at runtime based on the value assigned to it, and there is no need for explicit type declarations. This dynamic nature provides flexibility but can lead to errors if incompatible types are combined, such as adding a string and an integer without explicit conversion. Python’s type system includes a variety of built-in types like integers, floats, strings, and collections, as well as user-defined types for handling complex data.

Variables in Python are named references to data stored in memory. They have four essential properties: a name, which serves as an identifier; a value, which represents the data they hold; a type, which dictates the operations allowed; and a memory location, which is managed by Python's runtime system. For example, a variable x can hold an integer, a string, or a list, and its type can be checked with the type() function. Each variable’s memory address can also be inspected using the id() function.

Reassigning a variable means assigning it a new value, potentially changing its type. For instance, a variable holding an integer can be reassigned to a string or a list. While this is convenient, it can lead to risks like losing the original value or causing errors if the reassigned type doesn’t support certain operations. For example, reassigning a list variable to an integer removes its original list operations, potentially causing runtime errors when those operations are attempted. Reassignment is best used when a variable logically represents evolving data, such as counters in loops or accumulators in calculations. To mitigate risks, it is advisable to use new variables for clarity and avoid unnecessary type changes.

Understanding these concepts is essential for writing robust Python code. Mastery of types, variables, and their interactions enables developers to manage data effectively while reducing errors and maintaining readability.

## Exercises

1. Write a Python program that declares a variable, assigns it a value, and prints its value, type, and memory location using the type() and id() functions.
2. Create a variable and assign it three different types of values in sequence. Print the type and value of the variable after each assignment.
3. Declare variables of type integer, float, string, and boolean. Perform at least one operation with each type (e.g., addition for numbers, concatenation for strings). Print the results.
4. Write a function that accepts a variable and prints whether it is an integer, string, float, or boolean. Use the isinstance() function for type checking.
5. Assign an integer to a variable and convert it to a float and a string. Then assign a string containing a number and convert it back to an integer. Print the results of each conversion.
6. Declare a variable and assign it a string value. Reassign it to an integer and then to a float. Attempt to perform an operation that is valid for strings but invalid for numbers, and observe the resulting error.
7. Assign a value to a variable and print its memory address using id(). Reassign a new value to the variable and print the updated memory address. Compare the addresses to see if they change.
8. Create a program that declares a variable with a numeric value. Check if the variable is an integer or a float using the isinstance() function. Then, convert it to the other numeric type (int to float or float to int) and check its type again. Print the results.
9. Write a Python program that declares a variable with an initial value. Print its type and memory address. Then, assign the same value to another variable and compare their memory addresses using id(). Explain whether Python creates a new object or reuses the existing one based on the type of the value.
10. Declare a variable with a boolean value. Convert it to an integer, a float, and a string. Print the results of each conversion and explain the output.