Variables and Data Types in Python

In Python, a **variable** is a name that refers to a value stored in memoryrealpython.com. Variables themselves have no fixed type; instead, they reference objects of some type. When you assign a value to a variable, Python creates an object of the appropriate type and makes the variable name point to it. For example:

```
x = 5           # x refers to an integer object (int)
y = "Alice"     # y refers to a string object (str)
```

Here, `x` holds an integer object with value `5`, and `y` holds a string object `"Alice"`. Because Python is **dynamically typed**, you never declare a variable's type ahead of time. The interpreter infers the type from the assigned valuerealpython.comgeeksforgeeks.org. For instance, you can later reassign `x` to a different type:

```
x = 10          # x is now int
x = "hello"     # x is now str (dynamic typing)
```

Python variables follow simple naming rulesgeeksforgeeks.orgrealpython.com:

- Names can contain letters (A–Z, a–z), digits (0–9) and underscores (_).
- They **cannot start with a digit** (e.g. `1var` is invalid).
- Names are **case-sensitive** (`myVar` and `myvar` are different)realpython.com.
- Avoid using Python's reserved keywords (e.g. `if`, `for`, `class`) as variable namesgeeksforgeeks.org.

For example, valid names include `_count`, `total_score`, or `user1`; invalid names would be `123name` or using a keyword like `class`.

Because Python infers types at runtime, a variable's type is determined by the object it points torealpython.com. You can check an object's type with the built-in `type()` function:

```
name = "Jane"
age = 30
fruits = ["apple", "banana"]
print(type(name))    # <class 'str'>
print(type(age))     # <class 'int'>
print(type(fruits))  # <class 'list'>
```

In these examples, `type(name)` is `str`, `type(age)` is `int`, and `type(fruits)` is `list`realpython.com. Note that *variables* don't themselves have a fixed type – only the objects dorealpython.com.

Python Data Types

Python classifies values by **data type**. Since everything is an object in Python, each built-in data type is actually a class, and variables hold instances of these classesgeeksforgeeks.org. The principal built-in types include:

- **Numeric types** (integers, floating-point, complex)
- **Sequence types** (strings, lists, tuples, ranges)
- **Mapping types** (dictionaries)
- **Set types** (sets, frozensets)
- **Boolean type** (`bool`)
- **Binary types** (bytes, bytearray, memoryview)

Below we describe each major type with its characteristics and usage.

Numeric Types: `int`, `float`, `complex`

Numeric types represent numbersgeeksforgeeks.orgdocs.python.org.

- **Integers** (`int`): Whole numbers, positive or negative, with no fractional part. Python integers have **unlimited precision** (limited only by memory)geeksforgeeks.orgdocs.python.org. For example, `42` or `-1000000000000000000` are `int`.
- **Floating-point numbers** (`float`): Real numbers with a decimal point or in scientific notation. These are usually double precision (about 15–17 decimal digits) under the hood. Examples: `3.14`, `-0.001`, or `1e9`.
- **Complex numbers** (`complex`): Numbers with real and imaginary parts, written as `a + bj`, where `a` and `b` are floats and `j` denotes the imaginary unit. For example, `2+3j` has real part 2 and imaginary part 3. Python provides a `complex` type for these.

Example usage:

```
a = 5           # int
b = 3.14        # float
c = 2 + 4j      # complex
print(type(a), type(b), type(c))
```

This would output `<class 'int'> <class 'float'> <class 'complex'>`, showing each typegeeksforgeeks.org. Note that Booleans (`True`, `False`) are a subtype of integers (essentially 1 and 0)docs.python.org.

Text Type: `str`

Strings (`str`) represent text. A Python string is a sequence of Unicode charactersgeeksforgeeks.org. String literals can be delimited by single quotes (`'...'`), double quotes (`"..."`), or triple quotes (`'''...'''` or `"""..."""`) for multi-line text. There is no separate “char” type – individual characters are just strings of length 1.

Example:

```
s = "Hello, World!"
print(type(s))      # <class 'str'>
print(s[7:12])      # "World"
```

Strings are **immutable**, meaning you cannot change a character in place. Operations like concatenation or slicing always create new string objects. (For example, `"hello".upper()` returns a new string `"HELLO"`realpython.com.)

Boolean Type: `bool`

The Boolean type (`bool`) has only two possible values: `True` or `False`. Internally, `bool` is a subtype of `int` (where `True` acts like `1` and `False` like `0`), but it's a distinct type for claritydocs.python.orgtutorialspoint.com.

Example:

```
flag = True
print(type(flag))  # <class 'bool'>
print(1 + True)    # 2, since True == 1
```

Boolean values often arise from comparisons (`5 > 3` evaluates to `True`) or logical operations, and they can control flow in `if` statements and loops.

Sequence Types: `list` and `tuple`

Sequences are ordered collections of values. Common sequence types are lists and tuples.

- **Lists** (`list`): An ordered, mutable collection of items. Lists can hold items of any type, even mixed together. You create a list with square brackets:

  ```
  fruits = ["apple", "banana", "cherry"]
  print(type(fruits))  # <class 'list'>
  ```

  Lists are **mutable**, so you can change, add, or remove items after creation:

  ```
  fruits.append("date")
  fruits[1] = "blueberry"
  print(fruits)  # e.g. ['apple', 'blueberry', 'cherry', 'date']
  ```

  Unlike arrays in some languages, Python lists can contain heterogeneous types. The order of items is preserved, and you can index or slice lists by position (including negative indices for counting from the end).

- **Tuples** (`tuple`): An ordered collection of items **similar to lists but immutable**geeksforgeeks.org. You create a tuple with parentheses (or just commas):

```
coord = (10, 20)        # tuple of two ints
point = 1, 2, 3         # also a tuple of three ints (parentheses
optional)
print(type(coord))      # <class 'tuple'>
```

Once a tuple is created, its contents cannot be changedgeeksforgeeks.org. This immutability means tuples can be used as dictionary keys or set elements (because they are hashable)docs.python.org. Tuples are typically used for fixed collections of items, like coordinates or records.

Set Type: `set`

A **set** is an **unordered** collection of unique elementsgeeksforgeeks.org. Sets are defined by curly braces or the `set()` constructor. For example:

```
languages = {"Python", "Java", "C++"}
print(type(languages))  # <class 'set'>
```

Key characteristics of sets:

- **No duplicates**: A set automatically removes duplicate entries. For example, `{"apple", "banana", "apple"}` will only keep `"apple"` once.
- **Unordered**: Items in a set have no fixed order; iterating over a set may yield elements in any order.
- **Mutable**: You can add or remove elements with methods like `add()` or `remove()`. For example, `languages.add("Go")` updates the set in place.
- **Elements must be immutable**: Only hashable (immutable) objects can be stored as set elements (e.g. numbers, strings, tuples)tutorialspoint.com. For example, you cannot put a list or dict inside a set (it would raise a `TypeError`).

Example:

```
nums = {1, 2, 3}
nums.add(4)
nums.remove(2)
print(nums)  # e.g. {1, 3, 4}
```

Sets are useful for membership testing and eliminating duplicates, as well as mathematical set operations (union, intersection, etc.).

Mapping Type: `dict`

A **dictionary** (`dict`) is an **unordered collection of key–value pairs**geeksforgeeks.org. Think of it like a map or hash table. Each item in a dict has a **key** and a **value**, written as `key: value`. For example:

```
person = {"name": "Alice", "age": 30}
print(type(person))   # <class 'dict'>
```

Key properties of dictionaries:

- **Key–value pairs**: Each key is unique and maps to a value. E.g. `person["name"]` gives `"Alice"`.
- **Keys must be immutable**: You can use strings, numbers, or tuples as keys, but not mutable types like listsgeeksforgeeks.org. Keys are case-sensitive and unique.
- **Values can be any type**: Different key–value pairs can store different data types.
- **Mutable**: You can add, change, or delete entries. For example, `person["city"] = "London"` adds a new key.

Example:

```
d = {}
d["one"] = 1
d[2] = "two"
print(d)           # {'one': 1, 2: 'two'}
print(d.keys())    # dict_keys(['one', 2])
```

Dictionaries are not indexed by position, so you cannot slice them like a list. They are optimized for fast lookup by keygeeksforgeeks.orggeeksforgeeks.org.

Comparison of Data Types

Type	Example	Mutable?	Description
int	`x = 5`	No	Integer numbers (whole numbers, unlimited precision)geeksforgeeks.org
float	`x = 3.14`	No	Floating-point numbers (real numbers with decimal points)
complex	`x = 2+4j`	No	Complex numbers with real and imaginary partsgeeksforgeeks.org
str	`s = "hello"`	No	Text string (sequence of characters, Unicode)geeksforgeeks.org
bool	`flag = True`	No	Boolean value (`True` or `False`, subtype of int)geeksforgeeks.org
list	`lst = [1, "a", 3.0]`	**Yes**	Ordered sequence, can hold mixed types (mutable)geeksforgeeks.org
tuple	`t = (1, 2, 3)`	No	Ordered sequence, like list but **immutable**geeksforgeeks.org
set	`s = {1, 2, 3}`	**Yes**	Unordered collection of unique elements (mutable)geeksforgeeks.org
dict	`d = {"k": "v"}`	**Yes**	Mapping of keys to values (keys immutable, dict mutable)geeksforgeeks.orggeeksforgeeks.org

(In the table, "Mutable?" indicates whether you can change the object's contents after creation.)

Mutable vs. Immutable Types

A key distinction in Python is **mutability**. **Mutable** objects can be changed in place after they are created, whereas **immutable** objects cannot. For example, lists and dictionaries are mutable – you can add, remove or replace elementsrealpython.com. In contrast, numbers, strings, and tuples are immutable – any "change" to such objects actually creates a new objectrealpython.comgeeksforgeeks.org.

- **Mutable types:** `list`, `dict`, `set`, `bytearray`, etc. You can do `my_list.append(4)` or `my_dict["key"] = "value"` to change the object in place.
- **Immutable types:** `int`, `float`, `bool`, `str`, `tuple`, `frozenset`, etc. If you try to modify them, Python creates a new object. For example, concatenating strings `s = s + "!"` creates a new string; it doesn't change the original string.

Because immutable types cannot change, they can be **hashed** and used as dictionary keys or set elementsdocs.python.org. For example, you can use a tuple as a key but not a list. Real Python summarizes this distinction: "Mutable objects, such as lists and dicts, allow you to change their data directly… Immutable objects, like tuples and strings, don't allow in-place modifications"realpython.com.

Understanding which types are mutable is important when writing code. In general, **immutable** types tend to be simpler and safer (they can be used as keys, and you don't have to worry about unexpected modifications), whereas **mutable** types are more flexible for building and updating collections of data. Always keep mutability in mind when passing objects to functions or storing them in data structures.

Comparison: Lists vs. Tuples – a classic example. Lists (`[1,2,3]`) are mutable, so `lst[0] = 10` is allowed. Tuples (`(1,2,3)`) are immutable, so attempting `tup[0] = 10` raises an error. Real Python notes that mutable objects "allow you to change, add, or remove elements," whereas immutable ones "don't allow in-place modifications"realpython.com.

Overall, Python's dynamic typing and rich data types make it easy to work with data. Variables are simply labels attached to objects, and these objects can be of any built-in type (or user-defined class). The table and examples above summarize the common built-in types and their properties.