Python Primitive Data Types

Table of contents

Booleans 1
Integers 3
Strings 3
Tuples 10

The types "built-in" to Python by default are called **primitive data**: booleans, integers, floats, strings, and tuples. All of them are **immutable** — once created, an object of one of these types cannot be changed.

Booleans

The boolean type provides, comprehensively, the values `True` and `False`:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

Comparison operators

Having a boolean type is what makes **comparison operators** useful — they return a true/false answer:

```
>>> 3 == 9
False
>>> 3 > 7
False
>>> 3 >= 7
False
```

All comparison operators have **equivalent precedence** and are **left associative**; when mixed with arithmetic, comparison is evaluated *last* (see python-operator-precedence-and-associativity[1]).

Operator	Description
`==`	Equal
`!=`	Not equal
`<`, `<=`	Less than, less than or equal
`>`, `>=`	Greater than, greater than or equal

```
>>> 3 + 2 > 1 + 3   # arithmetic first
True
>>> 3+2 > 1+3       # better to group like this
True
>>> 3 + (2>1) + 3   # True has int value 1
7
```

Logical connectors

Functions that return booleans are called **predicates**, and are (very) important to programming. More sophisticated predicates are built with the logical connectors `and`, `or`, and `not` — covered in detail in 2025-08-11-sequence-selection-and-iteration[2] and python-boolean-logic[3].

`bool` — truthy / falsy

Any object can be converted to a boolean with `bool`. Data that converts to `True` is **truthy**; data that converts to `False` is **falsy**. Usually the falsy element is whatever acts as zero in the type, and everything else is truthy.

```
>>> bool(0)
False
>>> bool(1)
True
```

[1] courses/csse1001/python-operator-precedence-and-associativity.pdf
[2] courses/csse1001/2025-08-11-sequence-selection-and-iteration.pdf
[3] courses/csse1001/python-boolean-logic.pdf

Integers

> **Definition (Integer).** An **integer** is a number *without* a fractional part. It can be zero, positive, or negative.

Integers are **unbounded** in Python: we can work with integers of arbitrary size without worrying about **overflow**, which is not typical of programming languages.

```
>>> 2 ** 256 - 1
115792089237316195423570985008687907853269984665640564039457584007913129639935
```

Any object can be converted to an integer with `int`. `True` behaves as `1` and `False` as `0`, so booleans can be used in arithmetic:

```
>>> int(True)
1
>>> int(False)
0
>>> True + False
1
>>> True * False
0
```

Strings

> **Definition (String).** Anything (with some exceptions) enclosed by single-quotes `' '` or double-quotes `" "` is considered a **string** by Python. A string is an **ordered collection** of the characters (e.g. unicode and ascii) allowed by the computer.

The quotes are what make it a string; without them the same text is either a syntax error or a name:

```
>>> "hello world"
'hello world'
>>> type("hello world")
<class 'str'>
>>> hello world   # note the lack of quotes
SyntaxError: invalid syntax
>>> hello         # note the lack of quotes
NameError: name 'hello' is not defined
```

`str` conversion and formatting

Any object can be converted to a string with `str`:

```
>>> str(1)
'1'
>>> str(True)
'True'
>>> str("Hello")
'Hello'
```

Substitution converts values into strings inside a sentence, using the string formatter — an `f` before the quotes, and `{}` around each expression to substitute:

```
>>> x = 1
>>> y = "two"
>>> z = f"x is {x} y is {y}"     # formatted string
>>> z
'x is 1 y is two'
>>> print(f"x is {x} y is {y}")
x is 1 y is two
>>> f"{x}"                       # alternative to str
'1'
```

Concatenation and scalar multiplication

Adding strings concatenates them — and because a string can be added to itself repeatedly, it also makes sense to multiply a string by a positive integer:

```
>>> "hello" + "world"
'helloworld'
>>> type(" ")
<class 'str'>
>>> space = " "
>>> "hello" + space + "world"
'hello world'
>>> "Hello World!" + "Hello World!" + "Hello World!"
'Hello World!Hello World!Hello World!'
>>> 3 * "Hello World!"
'Hello World!Hello World!Hello World!'
```

When strings are added a **new string is created** — the originals are untouched, which is what immutability forces.

Comparing strings

Strings compare by character order; `ord` gives a character's order and `chr` recovers the character from the order:

```
>>> "a" < "b"
True
>>> ord("a"), ord("b")   # the "order" of the characters
(97, 98)
>>> chr(97)              # get character from order
'a'
>>> "A" < "a"
True
>>> "Z" < "a"
True
```

Comparison then proceeds character-by-character, with a shorter string counting as less than any extension of itself:

```
>>> "a" < "aa"
True
>>> "b" < "aa"
False
>>> "aba" < "ab"
False
>>> "aZ" < "aa"
True
```

> **Exercise.** Write `string_less_than(cs: str, ds: str) -> bool`, returning `True` only when `cs < ds`, restricting yourself to comparing integers only. `string_less_than("aba", "ab")` is false and `string_less_than("aZ", "aa")` is true.

Escape characters

A **new line** is an escape character that prints whatever follows it on a new line; the escape character is `\n`. A **tab** (`\t`) is a fixed amount of horizontal space, though how it is displayed depends on the program doing the displaying — which is why tabs are the worst.

```
>>> "hello\nworld"
'hello\nworld'
>>> print("hello\nworld")
hello
world
>>> "hello\tworld"
'hello\tworld'
>>> print("hello\tworld")
hello   world
```

A string can be **stored** differently than it is **printed**.

Numbers versus strings

`+` never silently converts between `int` and `str` — mixing them is a `TypeError`, and you have to say which conversion you meant:

```
>>> 3 + 7
10
>>> "3" + "7"
'37'
>>> 3 + "7"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> str(3) + "7"
'37'
>>> 3 + int("7")   # convert string to integer
10
```

Converting a string to a number only works for numbers expressed with digits, and `int` will not take a fractional literal:

```
>>> int("3.14")
ValueError: invalid literal for int() with base 10: '3.14'
>>> float("123.456")
123.456
>>> int("seven")
ValueError: invalid literal for int() with base 10: 'seven'
```

Length and inclusion

A string's **length** is the number of characters that comprise it:

```
>>> len("h")
1
>>> len("hello")
5
>>> cs = "world"
>>> len(cs)
5
>>> len(cs+"world") == len(cs) + len("world")
True
```

We can ask whether a string is a **substring** of another with `in`:

```
>>> "h" in "hello world"
True
>>> "hello" in "hello world"
True
>>> cs = "world"
>>> cs in "hello world"
True
>>> "ow" in "hello world"
False
```

Indexing

Because a string is *ordered*, we can number its characters **starting from zero** and access them with square brackets. We can also index from the end with negative numbers:

```
>>> cs = "hello world"
>>> cs[0]
'h'
>>> cs[1]
'e'
>>> cs[2]
'l'
>>> cs[len(cs)]
IndexError: string index out of range
>>> cs[-1]
```

```
'd'
>>> cs[-2]
'l'
>>> cs[-11]
'h'
```

Slicing

A slice `cs[start:stop:step]` takes *start* inclusive through *stop* exclusive, moving by *step*. The examples use `cs = "0123456789"` throughout, so that each index matches the character sitting at it.

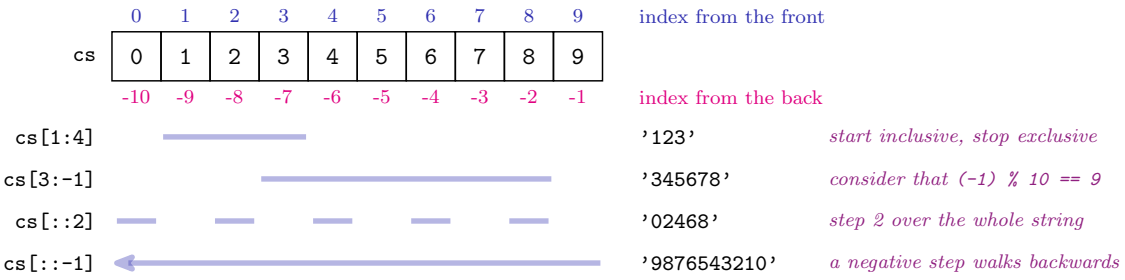

Figure 1: The string `cs = '0123456789'` with both index systems and four of its slices. Positive indices count from the front, negative from the back, and the two agree because a negative index i names position $i \bmod 10$ — which is why `cs[3:-1]` stops at 8, one before position 9.

Omitting an endpoint defaults it to the far end in the direction of travel:

```
>>> cs = "0123456789"
>>> cs[1:4]      # 1st inclusive through 4th exclusive
'123'
>>> cs[0:9]
'012345678'
>>> cs[0:10]
'0123456789'
>>> cs[0:]       # omitting the right endpoint defaults to the line above
'0123456789'
>>> cs[:-1]      # omitting the left endpoint defaults to zero
'012345678'
>>> cs[:]        # omitting both gives everything
'0123456789'
```

Negative endpoints are the same positions counted from the back — read a negative index modulo the length:

```
>>> cs[-1] == cs[len(cs)-1]
True
>>> cs[3:-1]     # consider that (-1) % 10 == 9
'345678'
>>> cs[-7:]      # consider that (-7) % 10 == 3
'3456789'
```

A step may be given third, and a negative step travels backwards:

```
>>> cs[::2]      # increment by two, zero inclusive to ten exclusive
'02468'
>>> cs[0:10:2]   # the above defaulted to this
'02468'
>>> cs[::-1]     # decrement by one, ten inclusive to zero exclusive
'9876543210'
>>> cs[10::-1]   # the above defaulted to this
'9876543210'
>>> cs[:10:-1]   # not to this, because we are moving backwards
''
```

Mixing all three, forwards and backwards:

```
>>> cs[1:-3:3]    # increment by three, one inclusive to seven exclusive
'14'
>>> cs[-8:-2:2]   # increment by two, two inclusive to eight exclusive
'246'
>>> cs[-8:-2:-4]  # decrement by four, two inclusive to eight exclusive
''
>>> cs[::-4]      # decrement by four, ten inclusive to zero exclusive
'951'
```

Immutability

Something is **immutable** when it cannot be changed. **Strings are immutable** — so a word cannot be capitalised by writing over its first character:

```
>>> cs = "hello"
>>> cs[0] = "H"
TypeError: 'str' object does not support item assignment
```

Strings as booleans

The empty string is the falsy string; every other string is truthy. It is also "smaller" than every other character:

```
>>> bool("a")
True
>>> bool("Hello")
True
>>> bool("")
False
>>> "" < "A"
True
>>> "" < chr(0)
True
```

A space is not the same thing as empty:

```
>>> space = " "      # a space
>>> len(space)
1
>>> bool(space)
True
>>> empty = ""       # empty string
>>> len(empty)
0
>>> bool(empty)
False
```

Tuples

> **Definition (tuple).** A **tuple** is an *immutable* ordered collection of elements. These elements are *not necessarily* the same type and are *not* necessarily distinct. Round brackets `()` are used to create tuples in Python.

Being ordered, tuples index and slice exactly like strings; being immutable, they refuse item assignment exactly like strings:

```
>>> xs = (0, 1, 2, 3, 4, 5)
>>> type(xs)
<class 'tuple'>
```

```
>>> xs[-1]        # indexing
5
>>> xs[0:4]       # slicing
(0, 1, 2, 3)
>>> xs[0] = -1    # immutable
TypeError: 'tuple' object does not support item assignment
```

Tuples support the same addition and scalar multiplication as strings, and their elements can be of mixed type:

```
>>> xs = (1, 'a', 2, 'b')    # mixed types are fine
>>> ys = (3, 'c', 4, 'd')
>>> xs + ys                  # tuple addition
(1, 'a', 2, 'b', 3, 'c', 4, 'd')
>>> 2*xs                     # tuple scalar multiplication
(1, 'a', 2, 'b', 1, 'a', 2, 'b')
```

Nomenclature

Tuples are named by size: couple (2), triple (3), quadruple (4), quintuple (5), sextuple (6), septuple (7), octuple (8), and an n-tuple in general. The general form `(0, 1, 2, ..., n-1)` is *not* valid Python syntax — it is informal notation for the pattern.

Singleton and empty tuples

A single value in round brackets is **not** a tuple — brackets there are just grouping. The trailing comma is what makes a singleton:

```
>>> (1) + (2)
3
>>> type((1))
<class 'int'>
>>> (1)
1
>>> type((1,))
<class 'tuple'>
>>> (1,) + (2,)
(1, 2)
>>> (1) + (2,3)   # common error
TypeError: unsupported operand type(s)
```

The empty tuple is `()` — a genuine tuple, the identity for tuple addition, and falsy:

```
>>> ()
()
>>> type(())
<class 'tuple'>
>>> () + (1,)
(1,)
>>> (1,) + ()
(1,)
>>> bool(())
False
```

Comparing tuples

Tuples compare element-by-element, and — like strings — a shorter tuple is less than a longer tuple that extends it:

```
>>> (1, 2, 3) == (1, 2, 3)
True
>>> (1, 2, 3) == (1, 2)
False
>>> (1, 2, 3) < (1, 2, 4)
True
>>> (1, 2, 3) < (1, 2)
False
>>> (1, 2) < (1, 2, 3)
True
```

Packing and unpacking

Several names can be assigned on one line, and several values printed on one line, through an (implicit) tuple:

```
>>> x, y, z = 2, 3, 4    # saves vertical space
>>> x
2
>>> y
3
>>> z
```

```
4
>>> x, y, z              # repl-printing several values
2, 3, 4
```

A real REPL echoes the last line as the tuple it built, brackets included — `(2, 3, 4)` — which is the more useful way to read it: the right-hand side of a packing assignment is a tuple.