Python Primitive Data Types

Table of contents

The types "built-in" to Python by default are called **primitive data**: booleans, integers, floats (not covered here), strings, and tuples.

Booleans

The boolean type comprehensively provides the values `True` and `False` (`type(True)` and `type(False)` are both `bool`).

Comparison operators

All comparison operators have equal precedence and are left-associative; when mixed with arithmetic, comparison is evaluated *last*.

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**. More sophisticated predicates can be built with the logical connectors `and`, `or`, and `not` (covered in detail in the if-statement lecture).

`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 for the type, and everything else is truthy.

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

Integers

An **integer** is a number without a fractional part — zero, positive, or negative. Integers are unbounded in Python, so we can work with arbitrarily large integers without overflow (which is unusual amongst languages):

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

Booleans convert to integers with `int` (`True` behaves as `1`, `False` as `0`):

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

Strings

A **string** is (with some exceptions) anything enclosed by single- or double-quotes — an ordered collection of the characters (e.g. unicode/ascii) the computer allows.

```
>>> "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

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

Formatted strings (`f"..."`) substitute variables into a string:

```
>>> x = 1
>>> y = "two"
>>> f"x is {x} y is {y}"
'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 creates a new string; multiplying a string by a positive integer repeats it:

```
>>> "hello" + "world"
'helloworld'
>>> space = " "
>>> "hello" + space + "world"
```

```
'hello world'
>>> 3 * "Hello World!"
'Hello World!Hello World!Hello World!'
```

Comparing strings

Strings compare lexicographically by character order (`ord`/`chr` give a character's order and the character at an order):

```
>>> "a" < "b"
True
>>> ord("a"), ord("b")
(97, 98)
>>> chr(97)
'a'
>>> "A" < "a"
True
>>> "Z" < "a"
True
```

A shorter string is less than an extension of itself, but otherwise comparison proceeds character-by-character:

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

> The lecture previews an exercise (`string_less_than(cs, ds)`, restricted to comparing integer character codes) that it explicitly defers ("we will return to this") without giving a worked solution — flagged here rather than invented.

Escape characters

`\n` (new line) and `\t` (tab) are escape characters — a string can be *stored* differently than it is *printed*:

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

Tabs display as a fixed amount of horizontal space, but exactly how much depends on the program displaying them.

Numbers versus strings

`+` does not silently convert between `int` and `str` — mixing them raises a `TypeError`:

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

`int()`/`float()` only work on strings that are actually numbers expressed as digits:

```
>>> 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

```
>>> len("hello")
5
>>> cs = "world"
>>> len(cs+"world") == len(cs) + len("world")
True
>>> "h" in "hello world"
True
>>> "ow" in "hello world"
False
```

Indexing and slicing

Strings are ordered, so characters are numbered from zero and accessed with square brackets. Negative indices count from the end:

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

A slice `cs[start:stop:step]` grabs the *start*-inclusive, *stop*-exclusive characters, stepping by *step* (default 1); omitted endpoints default to the whole string in that direction, and a negative step reverses direction:

```
>>> cs = "0123456789"
>>> cs[1:4]
'123'
>>> cs[:-1]
'012345678'
>>> cs[::2]
'02468'
>>> cs[::-1]
'9876543210'
```

Immutability

Strings are **immutable** — they cannot be changed in place:

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

Strings as booleans

```
>>> bool("Hello")
True
>>> bool("")
False
```

The empty string is "smaller" than every other string under comparison (`"" < "A"` is `True`) — note it is distinct from a single space (`len(" ") == 1`, `bool(" ") == True`).

Tuples

A **tuple** is an immutable ordered collection of elements — elements need not share a type or be distinct. Round brackets `()` construct tuples.

```
>>> xs = (0, 1, 2, 3, 4, 5)
>>> type(xs)
<class 'tuple'>
>>> xs[-1]
5
>>> xs[0:4]
(0, 1, 2, 3)
>>> xs[0] = -1
TypeError: 'tuple' object does not support item assignment
```

Tuples support the same +/scalar-multiplication as strings:

```
>>> xs = (1, 'a', 2, 'b')
>>> ys = (3, 'c', 4, 'd')
>>> xs + ys
(1, 'a', 2, 'b', 3, 'c', 4, 'd')
>>> 2*xs
(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), … an n-tuple in general (`(0, 1, 2, ..., n-1)` isn't valid Python syntax itself — it's just informal notation for the pattern).

Singleton and empty tuples

A single value in round brackets *without* a trailing comma is not a tuple — it's just that value in parentheses. The trailing comma is what makes it a tuple:

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

The empty tuple `()` is falsy:

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

Comparing tuples

Tuples compare element-by-element, lexicographically (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, 4)
True
>>> (1, 2) < (1, 2, 3)
True
```

Packing and unpacking

Multiple assignment/printing in one line, via an (implicit) tuple:

```
>>> x, y, z = 2, 3, 4
>>> x, y, z
2, 3, 4
```