CSSE1001 — Week 2 Notes

Table of contents

Primitive Data

See csse1001[1] for course logistics — this note covers Lecture 2B's technical content.

[1] courses/csse1001/csse1001.pdf

Today's outline

- Primitive data types built into Python
- Booleans, integers, strings, and tuples
- Converting between types

Learning objectives

1. There are more built-in types besides integers that can form expressions.
2. These types are immutable.
3. We can convert between types, albeit imperfectly.

Primitive data

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

See python-primitive-data-types[2] for the full definitions, operators, and worked examples for each of these types — comparison operators and truthy/falsy values for booleans; unbounded-precision and conversion for integers; concatenation, comparison, indexing/slicing, and immutability for strings; and construction, indexing/slicing, immutability, and packing/unpacking for tuples.

Summary

We have booleans, integers, strings, and tuples. Strings and tuples are indexed and can be sliced. We can convert between the types (imperfectly).

Next lecture

1. 2025-08-11-functions[3] — Functions.
2. 2025-08-11-sequence-selection-and-iteration[4] — If-statements (selection).

[2] courses/csse1001/python-primitive-data-types.pdf
[3] courses/csse1001/2025-08-11-functions.pdf
[4] courses/csse1001/2025-08-11-sequence-selection-and-iteration.pdf

Python Memory Model

See csse1001[5] for course logistics — this note covers Lecture 2A's technical content.

Today's outline

- Memory as a sequence of bits
- Variables as named memory locations
- Sequencing: why the order of instructions matters

Learning objectives

1. Memory is a sequence of bits that we can toggle.
2. We can (and should) store values in named memory locations called variables.
3. The order of instructions matters.

How much can a byte hold?

Computer memory is a very long series of on/off switches (**bits**, short for *binary digit*). Eight consecutive bits form a **byte**, and 4 or 8 bytes form a **word** (depending on machine architecture).

In general, n bits can store 2^n distinct things:

- 1 bit: $2^1 = 2$ things (`off`, `on`)
- 2 bits: $2^2 = 4$ things (`off-off`, `on-off`, `off-on`, `on-on`)
- 3 bits: $2^3 = 8$ things
- 4 bits: $2^4 = 16$ things

Variables and memory

See python-variables-and-memory-model[6] for how Python interprets memory as typed objects, how `id()` reveals an object's address, and the rules for assigning, retrieving, naming, and reassigning variables.

[5] courses/csse1001/csse1001.pdf
[6] courses/csse1001/python-variables-and-memory-model.pdf

Sequencing: order matters

The same instructions executed in a different order will (usually) produce a different outcome.

Python Tutor 1 — note we *cannot* swap these two lines: `x = x + 1` on its own throws a `NameError`, since `x` has no value yet.

```
>>> x = 1
>>> x = x + 1
```

Python Tutor 2:

```
>>> x = 1
>>> x = x + 1
>>> x = 2*x
>>> x
4
```

Python Tutor 3 — last two lines swapped:

```
>>> x = 1
>>> x = 2*x
>>> x = x + 1
>>> x
3
```

Python Tutor 4:

```
>>> x = 2
>>> y = 3
>>> y = x
>>> x = y
>>> x
2
>>> y
2
```

Python Tutor 5 — middle two lines swapped:

```
>>> x = 2
>>> y = 3
>>> x = y
>>> y = x
>>> x
3
>>> y
3
```

Summary

We can manipulate memory using Python and refer to that memory with variables. The order in which we do this manipulation matters — the same sequence of instructions done in a different order will (usually) result in a different outcome.

Next lecture

2025-08-04-primitive-data[7] — immutable objects (the things we put in memory that cannot be changed).

Reference material

Python Primitive Data Types

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*.

[7] courses/csse1001/2025-08-04-primitive-data.pdf

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
```

Python Variables and Memory Model

Memory

Computer memory is a very long series of on/off switches, called **bits** (short for *binary digit*). Eight consecutive bits form a **byte**; 4 or 8 bytes form a **word** (depending on machine architecture).

We put something in memory by toggling the bits into some meaningful configuration — in Python, the "somethings" we put in memory are called **objects**.

Interpreting memory: the integer type

A word of memory can be interpreted as an arithmetic expression in binary — e.g. $1 \cdot 2^{32} + 1 \cdot 2^{31} + 1 \cdot 2^{30} + 0 \cdot 2^{29} + \cdots + 1 \cdot 2^0 = 3{,}931{,}377{,}233$.

When Python executes `x = 3931377233` it toggles the bits at `x`'s memory location to this value and remembers to interpret that region as an (unsigned 32-bit) integer — its **type**. A value's type ultimately dictates which functions are compatible with it.

Addresses and `id`

Memory is divided into addressed words. We can determine any object's address with `id`:

```
>>> 42
42
>>> id(42)
4384160760
```

Variables

A **variable** is a nickname we give to an address in memory.

Assigning

```
>>> x = 3931377233
>>>
```

Assignment toggles the bits at the memory location nicknamed `x`. Nothing is printed, because the imperative here is to perform an action (a state change to memory), not to evaluate an expression.

Retrieving

Once `x` has been *assigned* a value, we *retrieve* it by evaluating it in the REPL — a variable evaluates to the value it was assigned, and can stand in for that value inside larger expressions:

```
>>> x
3931377233
>>> x // 100
39313772
```

Evaluating a name that has never been assigned raises a `NameError`:

```
>>> bear
NameError: name 'bear' is not defined
```

Meaningful names

Variable names should convey meaning. `x = 2; y = 3; z = x*y` is not meaningful, but `width = 2; height = 3; area = width*height` is — and `area` keeps working even after `width`/`height` change, which is another reason we use variables that can vary.

Naming rules

Variable names must start with a letter (`a-z`, `A-Z`) or underscore (`_`). Starting a name with a digit is a `SyntaxError`; digits are otherwise allowed.

```
>>> 1 = "one"
SyntaxError: cannot assign to literal
>>> x1 = 1
>>>
```

Reassignment

= in Python means *assignment* ("gets"), not mathematical equality:

```
>>> x = 1         # x "gets" 1
>>> x = x + 1     # x "gets" x+1
>>> x
2
```

In mathematics, $x = x+1 \implies 0 = 1$, a contradiction — so this would be an "illegal" statement. In Python it's perfectly valid, because = is a one-time action performed left-to-right, not a persistent equality claim.