CSSE1001 — Week 2 Notes

Table of contents

Primitive Data **1**
- Outline . . . 2
- Learning objectives . . . 2
- Primitive data . . . 2
- Key results . . . 2
- Converting between types — "albeit imperfectly" . . . 3
- Exercise: string comparison . . . 3
- Summary . . . 4
- Next lecture . . . 4

Python Memory Model **4**
- Outline . . . 4
- Learning objectives . . . 4
- How much can a byte hold? . . . 5
- Variables and memory . . . 6
- Sequencing: order matters . . . 6
- Summary . . . 7
- Next lecture . . . 7

Reference material **9**
- Python Primitive Data Types . . . 9
- Python Variables and Memory Model . . . 20

Primitive Data

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

[1] courses/csse1001/csse1001.pdf

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**. They are:

- booleans,
- integers,
- float (which we skip),
- strings, and
- tuples.

Booleans, integers and strings each come with a conversion function that shares the type's name: `bool`, `int` and `str`.

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

Key results

A handful of results carry everything else:

- **Booleans are numbers when you need them to be.** `True` has integer value `1`, so `3 + (2>1) + 3` is `7`. Comparison has *lower* precedence than arithmetic, so `3 + 2 > 1 + 3` is `(3+2) > (1+3)`.
- **Falsy is whatever acts as zero.** `bool(0)`, `bool("")` and `bool(())` are all `False`; everything else in these types is truthy. A space is not empty — `len(" ")` is `1` and `bool(" ")` is `True`.

[2] courses/csse1001/python-primitive-data-types.pdf

- **Strings and tuples are the same shape.** Both are ordered, so both index from zero, index from the back with negatives, slice, concatenate with `+`, and repeat with `*`. Both are immutable, so both reject item assignment with the same `TypeError` — `'str' object does not support item assignment` and `'tuple' object does not support item assignment`. A tuple slices with the same `[start:stop:step]` syntax as a string, e.g. `xs[0:4]`.
- **The singleton tuple needs its comma.** `(1)` is the integer `1`; `(1,)` is the tuple. `(1) + (2,3)` is the common error.

Converting between types — "albeit imperfectly"

Each of `bool`, `int` and `str` doubles as a conversion function, which gives six conversions among those three types. Only one of the six can fail — which is what the third learning objective is about.

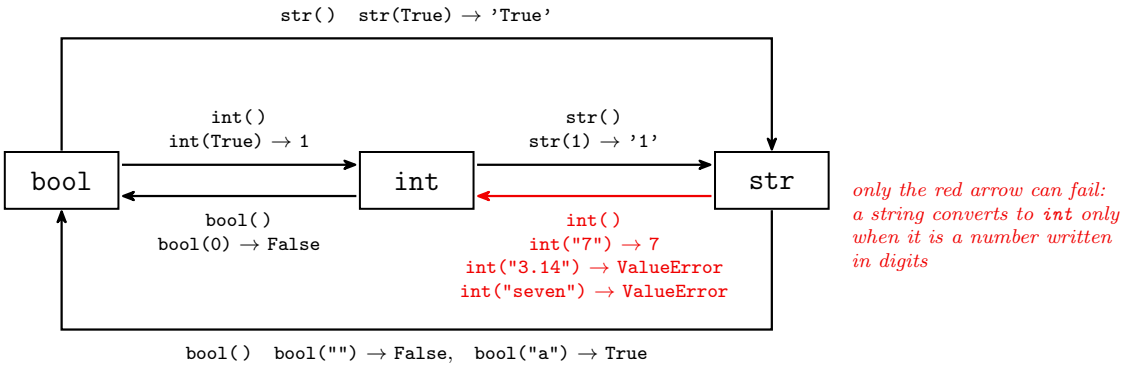

Figure 1: The six conversions among `bool`, `int` and `str`, each with an example call and its result. Only the red arrow, `str` to `int`, can fail. `float` is left out.

Python will not do the conversion for you: `3 + "7"` is a `TypeError`, and you have to say whether you meant `str(3) + "7"` (`'37'`) or `3 + int("7")` (10).

Exercise: string comparison

An exercise on string comparison:

```
def string_less_than(cs: str, ds: str) -> bool:
    """
    Restrict yourself to comparing integers only.
    Return true only when cs < ds.
```

```
>>> string_less_than("aba", "ab")
False
>>> string_less_than("aZ", "aa")
True
"""
```

"Restrict yourself to comparing integers only" rules out `<` on the strings themselves and points at `ord`. The two doctests are the two cases that catch a naive attempt — a prefix loses to its extension (`"aba" < "ab"` is `False`), and an uppercase letter beats a lowercase one, since `ord("Z")` is 90 and `ord("a")` is 97.

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

Python Memory Model

Lecture 2A. See csse1001[5] for course logistics.

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.

[3] courses/csse1001/2025-08-11-functions.pdf
[4] courses/csse1001/2025-08-11-sequence-selection-and-iteration.pdf
[5] courses/csse1001/csse1001.pdf

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).

To see how much can be stored in a byte, build up: one bit, two bits, three bits, four bits.

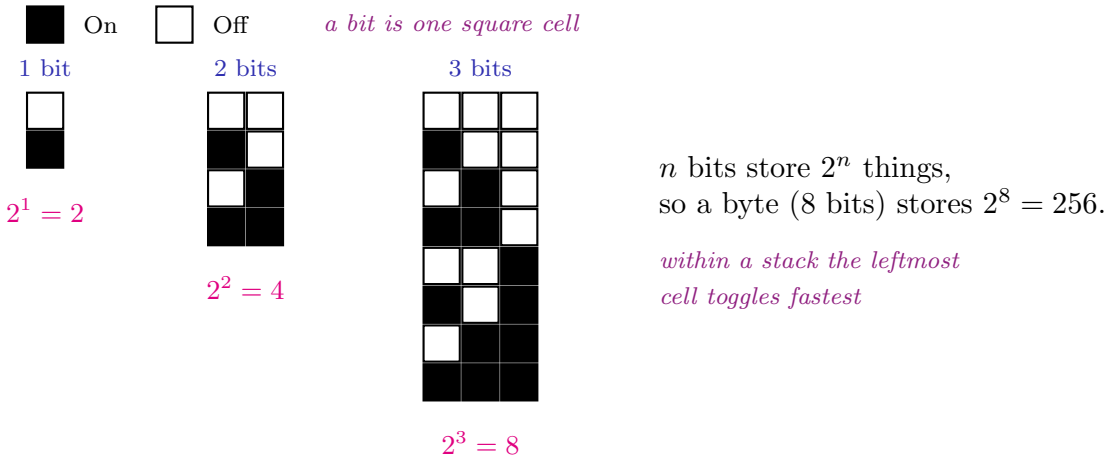

Figure 2: Adding a bit doubles the number of things you can code. The three stacks enumerate every pattern of one, two and three bits (filled = on, empty = off). Within each stack the **leftmost** cell toggles fastest (`off-off`, `on-off`, `off-on`, `on-on`), which reads the leftmost cell as the *least* significant; a word read as an integer is read the other way round. A byte is 8 bits, so it stores $2^8 = 256$ things.

In general, n bits 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

A byte is eight bits, so it codes $2^8 = 256$ things.

A very long series of such bits is computer memory. We *put something in memory* by toggling the on/off bits into some meaningful configuration; in Python the "somethings" we put in memory are called **objects**.

Variables and memory

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

- A word of bits means nothing until a **type** says how to read it — `x = 3931377233` both toggles the bits and records "read this as an unsigned 32-bit integer". The type dictates which functions are compatible with the object.
- Memory is divided into addressed words; `id(42)` returns the address where `42` is stored.
- A **variable** is a nickname for one of those addresses. Assignment prints nothing, because its imperative is to perform an action (a state change to memory).
- `=` is *assignment*, not equality: read `x = x + 1` as "`x` gets `x+1`".

Sequencing: order matters

The same instructions executed in a different order will (usually) produce a different outcome. Five short programs make the point, each one to step through in Python Tutor.

Python Tutor 1 — the two lines cannot be swapped:

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

`x = x + 1` cannot come first: the name `x` has no associated value on line 1, so evaluating the right-hand side throws a `NameError`.

Python Tutor 2 and **3** — the same three assignments, the last two swapped, giving 4 and 3:

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

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

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

Python Tutor 4 and **5** — two variables, and the assignment that gets clobbered decides which value survives:

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

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

The last two lines are the ones that change between these two examples — the first two assignments are identical.

Neither ordering swaps `x` and `y`, which is what these four lines look like they should do. Swapping two names needs a third name to hold one value while the other is overwritten (or the tuple packing of 2025-08-04-primitive-data[7]).

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[8] — immutable objects (the things we put in memory that cannot be changed).

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

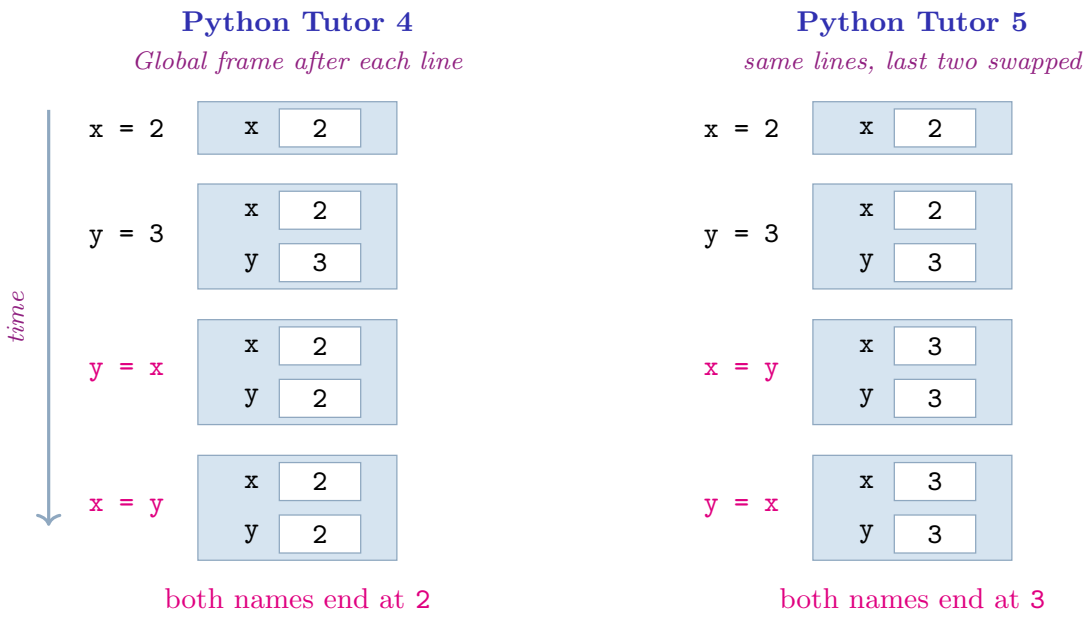

Figure 3: Why examples 4 and 5 disagree: the first assignment of the swapped pair overwrites one name, and the second then copies the *overwritten* value back — so both names always end up equal, but at 2 when `y = x` goes first and at 3 when `x = y` does. The two reordered lines are magenta. Each pale blue box is the global frame after that line, drawn the way Python Tutor shows it: one row per name, value in the slot. `y` does not exist until line 2.

Reference material

Python Primitive Data Types

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[9]).

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

[9] courses/csse1001/python-operator-precedence-and-associativity.pdf

```
>>> 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[10] and python-boolean-logic[11].

`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
```

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:

[10] courses/csse1001/2025-08-11-sequence-selection-and-iteration.pdf
[11] courses/csse1001/python-boolean-logic.pdf

```
>>> 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.

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'
```

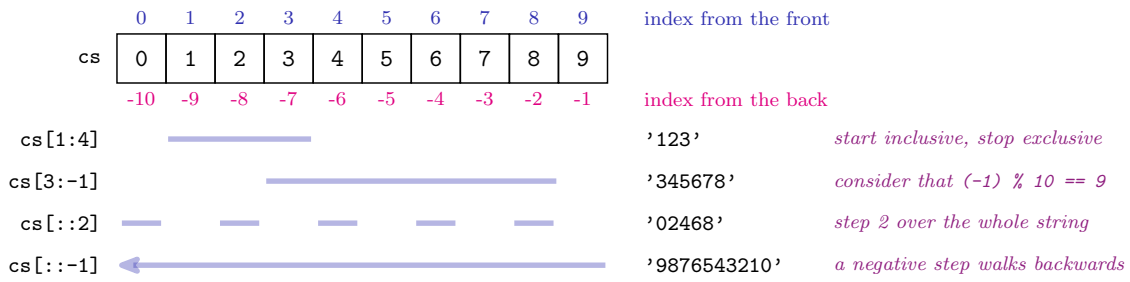

Figure 4: The string `cs = '0123456789'` with both index systems and four of its slices. Positive indices count from the front, negative from the back, and the two agree because a negative index i names position $i \bmod 10$ — which is why `cs[3:-1]` stops at 8, one before position 9.

```
>>> 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.

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).

A bit is drawn as one square cell: **filled black = on**, **white with a black outline = off**. A bit string is then a horizontal row of such cells read left to right, and memory is such rows

stacked vertically with an address in the left margin. Both figures below are drawn in that notation, which is collected in csse1001-diagram-notation[12].

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

The bits alone do not say what they mean. A word of memory can be *interpreted* as an arithmetic expression in binary — the place value of each position is a power of two, and the number is the sum of the place values where the bit is on.

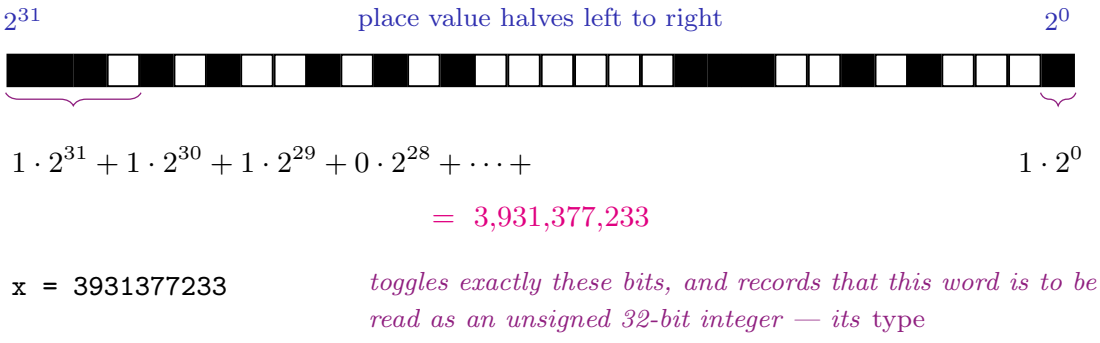

Figure 5: One word of memory read as an unsigned 32-bit integer (filled = on, empty = off). The 32 cells are exactly the binary expansion of 3,931,377,233 with the most significant bit on the left, so the positions run 2^{31} down to 2^0. The assignment `x = 3931377233` toggles exactly these bits and records that the word is to be read as an unsigned 32-bit integer. Nothing in the bits themselves says *integer* — that is what the type records.

When Python executes `x = 3931377233` it toggles the bits at `x`'s memory location to the above 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 the object.

Addresses and `id`

Memory is divided into words, and **each word gets an address**.

We can determine any object's address with `id`:

[12] courses/csse1001/csse1001-diagram-notation.pdf

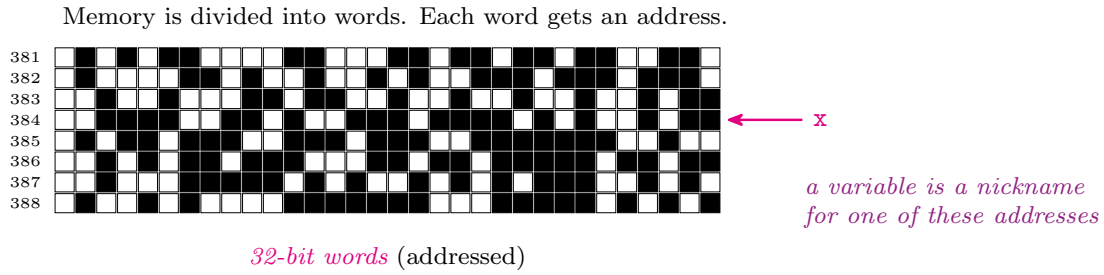

Figure 6: Memory as a stack of addressed 32-bit words, here at addresses 381–388. The bit contents are arbitrary. The magenta `x` and its arrow show that a variable is a nickname we give to an address in memory.

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

Simply using a number gets it loaded into memory, and `4384160760` is the memory location where 42 is stored.

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. This is not meaningful:

```
>>> x = 2
>>> y = 3
>>> z = x*y
```

These names do convey meaning:

```
>>> width = 2
>>> height = 3
>>> area = width*height
```

`area` keeps working even after `width` and `height` change — 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 inside a name.

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

The course marks against a fuller naming style: the PEP8 rule for variables and functions is lowercase, words separated by underscores, and class names are `CamelCaps`. See python-pep8-style-guide[13].

Reassignment

`=` in Python means *assignment*, not mathematical equality. Read `=` as "is assigned" or "gets":

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

In mathematics $x = x+1 \implies 0 = 1$, a contradiction ($\Rightarrow\Leftarrow$) — so as a mathematical statement it is illegal. In Python it is perfectly ordinary, because `=` performs a one-time action: evaluate the right-hand side, then toggle the bits at the location `x` names to that result.

Because assignment is an action, the **order** of assignments matters — see 2025-08-04-python-memory-model[14] for five worked examples.

[13] courses/csse1001/python-pep8-style-guide.pdf
[14] courses/csse1001/2025-08-04-python-memory-model.pdf