CSSE1001 — Week 4 Notes

Table of contents

Non-Primitive Data **2**
- Outline . . . 2
- Learning objectives . . . 2
- Lists . . . 2
- Dictionaries . . . 4
- Summary . . . 7
- Next lecture . . . 7

While-Loops **7**
- Outline . . . 7
- Learning objectives . . . 8
- Motivation . . . 8
- The two definitions . . . 8
- Accumulator pattern . . . 9
- When the condition never changes . . . 10
- Exercise: guess the number . . . 10
- Exercise: guess the dice roll . . . 11
- Further exercises . . . 12
- Summary . . . 12
- Next lecture . . . 12

Reference material **12**
- Python Dictionaries . . . 12
- Python Lists . . . 15
- Python Primitive Data Types . . . 22
- Python While Loops . . . 34

Non-Primitive Data

See csse1001[1] for course logistics. This is Lecture 4B.

Outline

- Lists: mutable ordered collections
- Dictionaries: hash tables

Learning objectives

1. Lists are mutable ordered collections; tuples (python-primitive-data-types[2]) are immutable ordered collections. Both can be sliced and indexed.
2. A name bound to a list is a *reference*, so copying, appending and passing to functions do not always do what they look like they do.
3. A dictionary maps keys to values via a hash function, and is unordered and mutable.

Lists

> **Definition (List).** A *list* is a *mutable* ordered collection of elements. These elements are *not necessarily* of the same type. Square brackets `[]` are used to create lists in Python.

See python-lists[3] for the full reference on lists: creation, mutability, comparison, membership, `append`/`extend`, aliasing, copying (shallow/deep), passing to functions, slicing, nested lists/matrices, type hints, and unpacking into function arguments. Two of those matter most and are drawn out there as figures — that `ys = xs` makes an *alias* rather than a copy, and that `.copy()` on a list-of-lists is only *shallow*.

Exercise: counting occurrences

```
def count(xs: list[int], ys: list[int]) -> int:
    """ Return the number of times members of ys are in xs.
    >>> count([1, 2], [1, 2, 3, 4])
    2
    >>> count([1, 2], [1, 2, 2, 2, 3, 1, 4])
    5
```

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

```
    """
```

A while-loop solution (python-while-loops[4]), walking `ys` by index:

```
def count(xs: list[int], ys: list[int]) -> int:
    ans, k = 0, 0
    while k < len(ys):
        y = ys[k]
        k += 1
        if y in xs:
            ans += 1
    return ans
```

It is certainly better to use for-loops in this situation:

```
def count(xs: list[int], ys: list[int]) -> int:
    ans = 0
    for y in ys:
        if y in xs:
            ans += 1
    return ans
```

There is also a one-line solution:

```
def count(xs: list[int], ys: list[int]) -> int:
    return sum(y in xs for y in ys)
```

(This last one relies on `True` counting as `1` under `sum`, and on the generator-expression syntax that 2025-08-25-comprehensions[5] introduces the week after.)

Exercise: rotating a list

We say the list `[0, 1, 2, 3, 4, 5]` *rotated* by 2 is `[2, 3, 4, 5, 0, 1]`. Write a function `def rotate(xs: list, k: int) -> list` that rotates a list by k.

This is left as an open exercise. Slicing (see python-lists[6]) is the hint about how it is meant to be solved.

[4] courses/csse1001/python-while-loops.pdf
[5] courses/csse1001/2025-08-25-comprehensions.pdf
[6] courses/csse1001/python-lists.pdf

Question: Caesar cipher

Write code for encrypting and decrypting messages using the Caesar cipher. In particular, write functions with the headers:

```
def encrypt_caeser(plaintext: str, shift: int) -> str:

def decrypt_caeser(ciphertext: str, shift: int) -> str:
```

This is left as an open exercise. A working `caesar_cipher` implementation (a single shift parameter instead of separate encrypt/decrypt functions) is in Lecture 5C — see week-five-exercises[7].

Dictionaries

Here is a deliberately vague question: write a function that returns the *character frequency* of a string, ignoring case. For example, the character frequency of `"hello world"` would encode that there is one "h", three " "'s and so on.

Viable representations

Two lists (characters paired positionally with their counts):

```
[['h', 'e', 'l', 'o', ' ', 'w', 'r', 'd'],
 [ 1,   1,   3,   2,   1,   1,   1,   1 ]]
```

One-to-one encoding (a count for every letter of the alphabet, in order):

```
# a b c d e f g h i j k l m n o p q r s t u v w x y z
 [0,0,0,1,1,0,0,1,0,0,0,3,0,0,2,0,0,1,0,0,0,0,1,0,0,0]
```

[7] courses/csse1001/week-five-exercises.pdf

Hash functions

In both cases we provided a way to map a **key** (character) to a **value** (number):

$$\texttt{str} \to \texttt{int}, \quad \texttt{h} \mapsto 1,\ \texttt{e} \mapsto 1,\ \texttt{l} \mapsto 3,\ \ldots$$

This mapping is called a **hash function**.

The hash function for the two-list encoding looks up the key's *position* in the first list, then indexes the second list at that position:

```
>>> def hash_1(key: str) -> int:
...     """Two list encoding."""
...     xs = ['h', 'e', 'l', 'o', ' ', 'w', 'r', 'd']
...     ys = [1, 1, 3, 2, 1, 1, 1, 1]
...     k = xs.index(key)   # position of key in xs
...     return ys[k]
>>> hash_1('o')
2
>>> hash_1('r')
1
```

The hash function for the one-to-one encoding computes the key's position in the alphabet directly, via `ord`:

```
>>> def hash_2(key: str) -> int:
...     """One-to-one encoding."""
...     xs = [0,0,0,1,1,0,0,1,0,0,0,3,0,0,2,0,0,1,0,0,0,0,1,0,0,0]
...     pos_in_alpha = ord(key) - ord('a')   # position of key in alphabet
...     return xs[pos_in_alpha]
>>> hash_2('o')
2
>>> hash_2('r')
1
```

Both return the same count for the letters of the example even though they store completely different things. That is the point of the exercise: the *encoding* is a free choice, and the hash function is whatever computation turns a key into an index into it.

They are not interchangeable in general. `hash_1` only knows the eight characters in its own `xs`, and `hash_2`'s `ord(key) - ord('a')` only lands inside its list for `'a'`–`'z'` — so the space character that `hash_1` counts is out of range for `hash_2`.

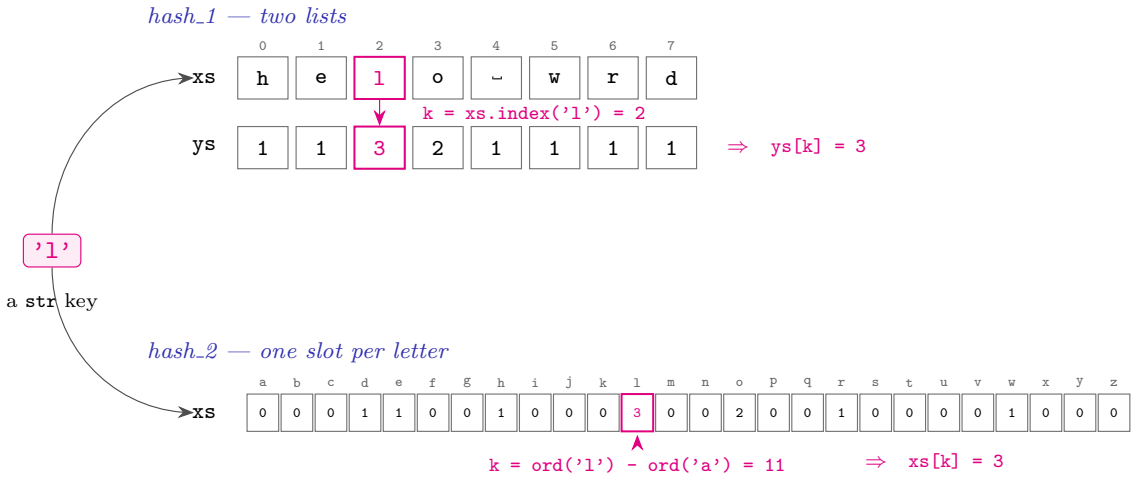

Figure 1: The same key routed through both encodings. `hash_1` finds the key's position in a list of characters and reads the count off a parallel list at that position; `hash_2` computes the position arithmetically from the character code and reads a single alphabet-sized list. Different storage, different index arithmetic, same answer — which is why Python's `dict` can hide the choice behind a *magic* hash function. Magenta marks the key and the value it leads to.

Generally speaking we can index a list by **any type of key** given some hash function which takes keys to values. Python has a *magic* hash function that will index anything appropriately. The result is implemented as the `dict` data-type.

Dictionaries have no ordering because the keys can be of mixed type.

The "hello world" example encoded using a `dictionary` is

```
>>> char_to_freq = {
...     'd': 1,
...     'o': 1,
...     ' ': 1,
...     'r': 1,
...     'w': 1,
...     'e': 1,
...     'l': 3,
...     'h': 0
...     }
```

> ⚠ Two of these counts are wrong
>
> Two entries disagree with the two-list encoding above (and with `"hello world"` itself): that encoding has `'o'` at **2** and `'h'` at **1**, which is what `hash_1('o')` returns above. As written, `char_to_freq` is wrong for the string it claims to encode.

See python-dictionaries[8] for the full reference on dictionaries: creation, indexing, `keys`/`values`/`items`, `clear`/`copy` (and its shallow-copy gotcha), `get`, and the requirement that keys be immutable.

Summary

Lists are mutable ordered collections; tuples are immutable ordered collections. Dictionaries are for storing *key-value* relationships — a key must be an *immutable* type and gets mapped to a value using a *hash function* that is magically fast.

Hash tables are a candidate for the most important/useful data-structure in computer science.

Next lecture

For-loops and comprehension — 2025-08-25-for-loops[9] and 2025-08-25-comprehensions[10].

While-Loops

See csse1001[11] for course logistics — this note covers Lecture 4A's technical content.

Outline

- Why we need to repeat code, possibly indefinitely
- The while-loop
- Simulating a do-while, and reading user input

[8] courses/csse1001/python-dictionaries.pdf
[9] courses/csse1001/2025-08-25-for-loops.pdf
[10] courses/csse1001/2025-08-25-comprehensions.pdf
[11] courses/csse1001/csse1001.pdf

Learning objectives

1. Loops are a control structure for repeating code.
2. A while-loop repeats code while a condition holds — and the condition is tested *before* each repetition.
3. `break` leaves a loop immediately; `while True` + `break` is how Python writes a do-while.
4. `input()` reads from the keyboard; `random.randint` generates random integers.

Motivation

There are (at least) two scenarios where *repeating possibly indefinitely* is **necessary**:

1. prompting the user for valid input, and
2. playing a random game (i.e. guessing dice rolls).

The common feature is that neither has a repetition count you could work out in advance, which is what a while-loop buys you over the counted `for`-loop of 2025-08-25-for-loops[12].

The two definitions

> **Definition (Loop).** A *loop* is a *control structure* that repeats code that belongs to it.
>
> **Definition (While-Loop).** A *while-loop* is a *control structure* that repeats code while some *condition is satisfied*.

In Python that is written

```
while <condition>:
    <code>
```

and it behaves like this:

Reading Figure 2, the only thing separating a while-loop from an `if` is the back-edge from the body to the diamond. See python-while-loops[13] for the full reference on loops, while-loops, `break`, augmented assignment operators (`+=`, `*=`, `/=`, `%=`), simulating a do-while, `input()`, and `random.randint`.

[12] courses/csse1001/2025-08-25-for-loops.pdf
[13] courses/csse1001/python-while-loops.pdf

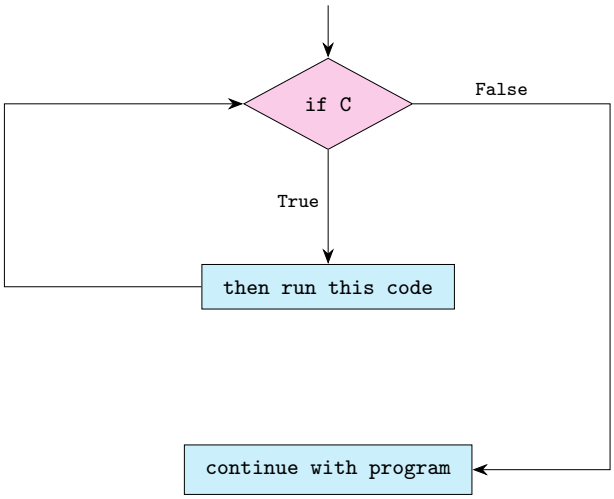

Figure 2: The while-loop as a flowchart. The condition `C` is tested *first*: while it is `True` control drops into the body and then loops straight back to the test, and only when it is `False` does control fall through to the rest of the program. The pink diamond is the decision and the blue boxes are code.

Accumulator pattern

The loop variable is updated inside the body so that the condition eventually fails:

```
>>> x = 0
>>> while x < 10:
...     x = x + 1          # accumulator
>>> x
10
```

Using an augmented assignment operator (see python-while-loops[14]) makes the accumulation more concise — `x += 1` is exactly `x = x + 1`:

```
>>> x = 0
>>> while x < 10:
...     x += 1             # accumulator
>>> x
10
```

The loop stops at exactly `10`, not `9`: the test `x < 10` is re-checked after the body has already pushed `x` to `10`.

[14] courses/csse1001/python-while-loops.pdf

When the condition never changes

If the condition is never falsified the loop never stops:

```
>>> x = 0
>>> while True:
...     print(x)
...     x += 1
0
1
2
3 ...
```

There is usually a key-stroke, typically `ctrl+c`, that terminates a loop — it is a good idea to learn what it is in your IDE.

The mirror case is a condition that is false to begin with. The body then never runs at all, because the test comes first:

```
>>> x = 0
>>> while False:
...     print(x)
...     x += 1
```

Nothing prints.

Exercise: guess the number

`x = input()` waits for input from the keyboard and then assigns it to `x` with `type(x) is str` — so a numeric guess has to be converted with `int(...)`.

Write a function `foo(target: int) -> int` that prompts the user to input a number until the user guesses some (secret) target. For each guess the function should print `Too high`, `Too low`, or `That's it!` depending on the guess. *Return the number of guesses required.*

```
>>> def foo(target: int) -> int:
...     number_of_guesses = 0
...     while True:
...         guess = int(input("Guess: "))
...         number_of_guesses += 1
...         if guess > target:
```

```
...             print("Too high")
...         elif guess < target:
...             print("Too low")
...         else:
...             print("That's it!")
...             return number_of_guesses
>>> foo(10)
Guess:
```

The loop is deliberately `while True`: there is no condition available at the top of the loop that says "the user has not guessed yet", because the guess does not exist until the body has run. The exit is the `return` in the `else` branch, which leaves the function (and so the loop) outright.

Exercise: guess the dice roll

Write a *function* called `bar() -> None` that prompts the user to predict the result of a six-sided dice roll. Repeat the prompt until the user predicts correctly.

Random number generation is handled by an *external library*. It is **not** built-in and therefore must be imported (see python-while-loops[15] for both import forms):

```
>>> from random import randint
>>> def bar() -> None:
...     """
...     Prompt user to predict the result of a six sided
...     dice roll until the guess is correct.
...     """
...     while True:
...         guess = int(input("Guess in [1..6]: "))
...         roll  = randint(1, 6)
...         if roll == guess:
...             break
...     return
>>> bar()
Guess in [1..6]:
```

Here the exit is a `break` rather than a `return`, because the function has nothing to hand back. The die is re-rolled every iteration, so the user is predicting a fresh roll each time rather than searching for one fixed hidden number.

[15] courses/csse1001/python-while-loops.pdf

Further exercises

Open exercises:

- Write a function that prints the result of a six-sided dice roll until a six is rolled, and returns the number of rolls required.
- Write a function that accumulates the result of dice rolls until some threshold is met/passed, and returns the number of rolls required.
- Develop functions for adding and multiplying positive integers together, restricting ourselves to exactly two arithmetic operations — `succ` (adding one) and `pred` (subtracting one). The two building blocks are:

```
def succ(x: int) -> int:
    return x+1

def pred(x: int) -> int:
    return x-1
```

Both the add and the multiply function are accumulators driven by a while-loop.

Summary

The *while-loop* is a control structure for repeating code a possibly infinite number of times.

Next lecture

2025-08-18-non-primitive-data[16] — lists and dictionaries.

Reference material

Python Dictionaries

> **Definition (Dictionary).** A *dictionary* in Python is a data-structure that stores `tuple(key, value)` pairings. Python has a "magic" hash function to find a key among the tuples fast. Dictionaries are **not ordered** and are **mutable**.

Hash tables are a candidate for the most important/useful data-structure in computer science.

The dictionary is not a new idea so much as a generalised index: see Lecture 4B's hash-function section[17] for two hand-rolled encodings, and the figure there showing a key being turned into

[16] courses/csse1001/2025-08-18-non-primitive-data.pdf
[17] courses/csse1001/2025-08-18-non-primitive-data.pdf

a list position by two different hash functions. `dict` is what you get when Python supplies that hash function for you, for keys of any (immutable) type. The reason a dictionary has no ordering is precisely that the keys can be of mixed type.

Creating and indexing

```
>>> h = {
...     "red": ["apple", "firetrucks", "cars"],
...     "yellow": ["banana", "cars"],
...     "blue": ["sky", "cars"]
...     }
>>> h["blue"]
['sky', 'cars']
>>> h["green"]
KeyError: 'green'
>>> h["green"] = ["leaves"]   # fine, because we're assigning not retrieving
```

An empty dictionary is `dict()` (or `{}`).

`keys`, `values`, `items`

```
>>> h.keys()
dict_keys(['red', 'yellow', 'blue'])
>>> h.values()
dict_values([['apple', 'firetrucks', 'cars'], ['banana', 'cars'], ['sky', 'cars']])
>>> h.items()
dict_items([('red', ['apple', 'firetrucks', 'cars']), ('yellow', ['banana', 'cars']), ('blue
```

`items()` is the one that shows the definition literally: each pairing comes back as a `(key, value)` tuple.

`clear` and `copy`

```
>>> f = h.copy()
>>> h.clear()
>>> h["blue"]
KeyError: 'blue'
>>> f["blue"]
['sky', 'cars']
```

Like list's `.copy()` (see python-lists[18]), dict's `.copy()` is only a **shallow copy** — mutable values are still shared:

```
>>> f = h.copy()
>>> h["blue"].append("windex")
>>> h["blue"]
['sky', 'cars', 'windex']
>>> f["blue"]
['sky', 'cars', 'windex']   # shallow copy -- same underlying list
```

Read the two transcripts together: `clear()` empties the dictionary `h` without touching `f`, because `f` is a different dictionary object — but `h["blue"].append(...)` reaches *through* both dictionaries to the one list they share. Same shallow/deep split as the shallow-copy figure in python-lists[19].

get

`.get(key)` is a safer alternative to `h[key]` — it returns `None` instead of raising `KeyError` if the key is missing:

```
>>> h.get("red")
['apple', 'firetrucks', 'cars']
>>> h["green"]
KeyError: 'green'
>>> h.get("green")
None                        # safer, because no error is thrown here
```

Without a method, the same "avoid a `KeyError`" pattern is written explicitly:

```
>>> if key in table:        # avoiding KeyError
...     table[key] += 1
... else:
...     table[key] = 0
```

This is a sketch of the membership-test idiom, not a working counter. As a *frequency counter* the `else` branch would have to be `table[key] = 1` — a key seen for the first time has been seen once, not zero times.

[18] courses/csse1001/python-lists.pdf
[19] courses/csse1001/python-lists.pdf

Warning: keys must be immutable

The **keys** of a dictionary **must** be an **immutable** type (see python-primitive-data-types[20] and python-lists[21]):

```
>>> d = dict()
>>> d[ [1,2,3] ] = 1
TypeError: unhashable type: 'list'
>>> d[ (1,2,3) ] = 1        # tuples are immutable
>>> h = dict()
>>> d[ h ] = 1
TypeError: unhashable type: 'dict'
```

"Unhashable" is the error message's way of saying the same thing the definition does: a key has to be something the hash function can turn into a fixed location, and a value that can change underneath you cannot be.

Python Lists

A **list** is a **mutable** ordered collection of elements. These elements are *not necessarily* of the same type. Square brackets `[]` are used to create lists in Python.

```
>>> xs = [1, "apple"]
>>> type(xs)
<class 'list'>
>>> xs[0]
1
>>> xs[0] = 2*xs[1]
>>> xs[0]                  # note: lists are mutable
'appleapple'
```

Comparison

Lists compare point-wise from position zero:

[20] courses/csse1001/python-primitive-data-types.pdf
[21] courses/csse1001/python-lists.pdf

```
>>> [1, 2, 3] < [4, 5, 6]
True
>>> [7, 2, 3] < [4, 5, 6]
False
>>> [] < [1]
True
```

The second line is the one to read carefully: the comparison stops at position zero, where `7 < 4` is `False`, and never looks at positions one and two.

Membership

```
>>> 1 in [1, 2, 3]
True
>>> 0 in [1, 2, 3]
False
>>> 123 in [1, 2, 3]
False
>>> [1] in [1, 2, 3]
False
```

`in` tests for an *element*, not a sub-list — the last line is `False` because `[1, 2, 3]` contains the integer `1`, not the list `[1]`.

Adding elements: `append`, `extend`, concatenation

`xs + [y]` **creates a new list**, which must be reassigned back to `xs` to have an effect:

```
>>> xs = [0, 1, 2]
>>> xs = xs + [3]
>>> xs
[0, 1, 2, 3]              # "this is the original list updated"
```

Read the comment as a statement about the *value* of `xs`, not about the object: `xs = xs + [...]` builds a **new** list and rebinds `xs` to it (see the aliasing transcript below), which is exactly why it breaks an alias where `append` does not.

Alternatively, using a method that `list` provides — `xs.append(y)` mutates `xs` **in place**:

```
>>> xs = [0, 1, 2]
>>> xs.append(3)
>>> xs
[0, 1, 2, 3]
```

`append` adds its argument as a *single new element*, even when that argument is itself a list:

```
>>> xs = [1, 2, 3]
>>> xs.append([4,5])
>>> xs
[1, 2, 3, [4, 5]]
```

`xs.extend(ys)` instead appends every element of `ys` individually — code similar to `xs = xs + ys`:

```
>>> xs = [1, 2, 3]
>>> xs.extend([4,5])
>>> xs
[1, 2, 3, 4, 5]
```

Aliasing

Take care when (trying to) copy lists. Assigning one list variable to another does **not** copy it — both names refer to the *same* list object:

```
>>> xs = [1, 2, 3]
>>> ys = xs
>>> ys[-1] = 9
>>> ys
[1, 2, 9]
>>> xs
[1, 2, 9]
```

Whether an operation preserves this aliasing relationship depends on whether it mutates the list in place or creates a new one:

```
>>> xs = [1, 2, 3]
>>> ys = xs           # ys is an alias of xs
>>> xs.append(4)      # appending to a list
>>> ys
```

```
[1, 2, 3, 4]
>>> xs += [5]          # appending to a list
>>> ys
[1, 2, 3, 4, 5]
>>> xs = xs + [6]      # concatenating two lists
>>> ys
[1, 2, 3, 4, 5]        # aliasing relationship broken!
```

The trap is that `xs += [5]` and `xs = xs + [6]` look like the same operation and are not: the first mutates the object both names share, the second builds a *new* object and rebinds only `xs`.

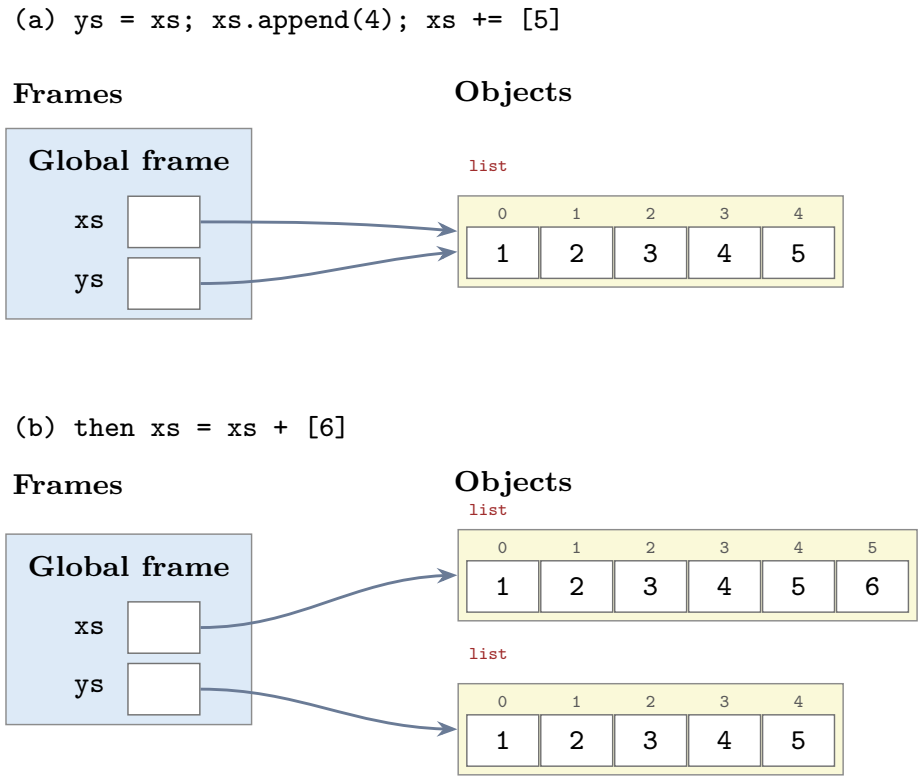

Figure 3: What the transcript above is doing to memory. In (a) `xs` and `ys` are two names for one list object, so the in-place operations `append` and `+=` are visible through both. In (b) `xs = xs + [6]` builds a second list object and rebinds `xs` to it; `ys` still refers to the original, which is what *aliasing relationship broken* means. The layout is Python Tutor's: frames on the left in pale blue, objects on the right in pale yellow, a reference as an arrow, and aliasing as two arrows converging on one object.

`.copy()` breaks the alias for a flat list:

```
>>> xs = [1, 2, 3]
>>> ys = xs.copy()
>>> ys[-1] = 9
>>> xs
[1, 2, 3]
>>> ys
[1, 2, 9]
```

Passing lists to functions

A more nuanced case is when passing lists (but really *list references*) to functions — mutating a parameter inside a function mutates the caller's list too:

```
>>> def foo(ys):
...     ys[0] = -100
>>> xs = [1, 2, 3]
>>> foo(xs)
>>> xs
[-100, 2, 3]
```

This is the same picture as (a) in Figure 3, with `ys` being a name in `foo`'s frame rather than the global one.

Nested lists and matrices

A list can have another list as an element:

```
>>> xs = [ [1,2], [5,6,7] ]
>>> len(xs)
2
>>> len(xs[-1])
3
```

`len` counts the *outer* elements — two, not five.

A **matrix** can therefore be represented by a list-of-lists in Python:

$$\mathbb{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \equiv \texttt{[[1, 2, 3], [4, 5, 6], [7, 8, 9]]}$$

```
>>> ass = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> ass[-1]
[7, 8, 9]
>>> ass[1:3]
[[4, 5, 6], [7, 8, 9]]
>>> ass[1]
[4, 5, 6]
>>> ass[1][-1]
6
>>> ass[1,-1]
TypeError: list indices must be integers or slices, not tuple
```

The last line is the one worth remembering: a nested list is indexed with *repeated* brackets, `ass[1][-1]`. The comma form `ass[1,-1]` is a tuple and is a `TypeError` for a list.

Deep copying

`.copy()` only performs a **shallow copy** — nested mutable elements (like inner lists) are still shared between the original and the copy:

```
>>> ass = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> bss = ass.copy()
>>> bss[0][0] = 0
>>> bss
[[0, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> ass
[[0, 2, 3], [4, 5, 6], [7, 8, 9]]   # A was changed!
```

A true independent copy of nested structures needs a **deep copy**, which is *not* built into `list` — see `copy.deepcopy`.

Slicing

```
>>> xs = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> xs[3:6]
[3, 4, 5]
>>> xs[::2]
[0, 2, 4, 6, 8]
>>> xs[7:2:-2]
[7, 5, 3]
```

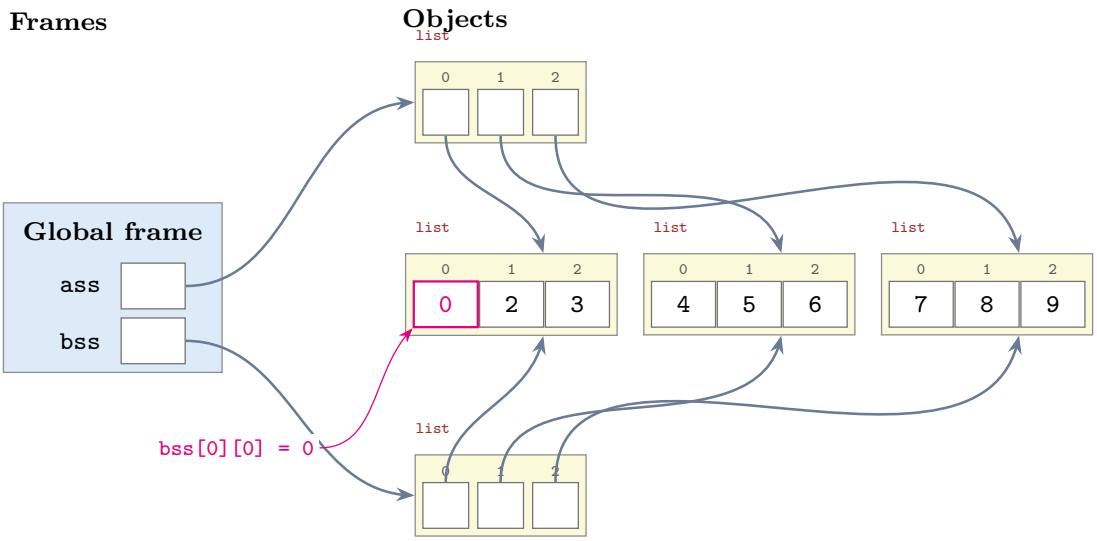

Figure 4: Why `ass` changed. `bss = ass.copy()` made a genuinely new *outer* list — that is why `bss` and `ass` are two distinct objects — but it copied the outer list's three references, not the objects they point at, so both outer lists share the same three inner lists. Writing `bss[0][0] = 0` reaches through a shared reference and the change shows up through `ass` as well. For this to work as expected we need a *deep-copy*, which is not built-in — see `copy.deepcopy`. Magenta marks the mutated cell.

The three fields are `start:stop:step` with `stop` exclusive: `xs[3:6]` takes positions 3, 4 and 5; `xs[::2]` takes every second element of the whole list; and a negative step makes `xs[7:2:-2]` walk *backwards* from position 7, stopping before position 2.

Type hints

> **Warning.** Type-hinting lists (e.g. `list[int]`) is a new feature in Python 3.9 and greater.

Unpacking into function arguments

It is possible to "unbracket" a list when passing it to functions, with `*`:

```
>>> def f(x, y, z):
...     return x + y + z
>>> x = [1, 2, 3]
>>> f(x)
TypeError: f() missing 2 required positional arguments: 'y' and 'z'
>>> f(*x)
6
```

Tuples

Tuples are **immutable ordered collections** whereas lists are **mutable ordered collections**. Both can be sliced and indexed. See python-primitive-data-types[22]; a tuple's immutability is what makes it usable as a dictionary key, where a list is not — see python-dictionaries[23].

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.

[22] courses/csse1001/python-primitive-data-types.pdf
[23] courses/csse1001/python-dictionaries.pdf

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[24]).

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
```

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

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[25] and python-boolean-logic[26].

`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:

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

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

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.

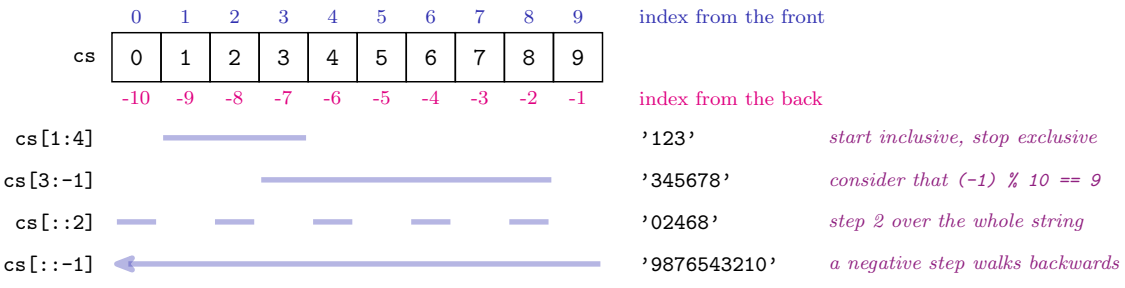

Figure 5: The string `cs = '0123456789'` with both index systems and four of its slices. Positive indices count from the front, negative from the back, and the two agree because a negative index i names position $i \bmod 10$ — which is why `cs[3:-1]` stops at 8, one before position 9.

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.

Python While Loops

A **loop** is a control structure that repeats code that belongs to it. A **while-loop** is a loop that repeats code *while some condition is satisfied*:

```
while <condition>:
    <code>
```

The condition is checked before every repetition (including the first) — if it's false to begin with, the loop body never runs at all:

```
>>> x = 0
>>> while False:
...     print(x)
...     x += 1
>>> # nothing prints
```

See the while-loop control-flow figure in 2025-08-18-while-loops[27] for a picture of this.

[27] courses/csse1001/2025-08-18-while-loops.pdf

Augmented assignment operators

Operator	Equivalent to
`x += y`	`x = x + y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`
`x %= y`	`x = x % y`

These are the four used in the course. They are what makes an accumulator loop readable:

```
>>> x = 0
>>> while x < 10:
...     x += 1
>>> x
10
```

`break`

The `break` keyword terminates the (innermost) loop immediately and continues with the rest of the program — so code after it *inside the loop body* is unreachable:

```
>>> while True:
...     print("hello")
...     break
...     print("world")
hello
```

Simulating a do-while

A **do-while** (or **repeat-until**) is a while-loop variant available in other languages but not natively supported in Python. It runs its code *at least once*, because the condition is only tested after the body:

```
do:
    <code>
while <condition>   # not valid Python syntax
```

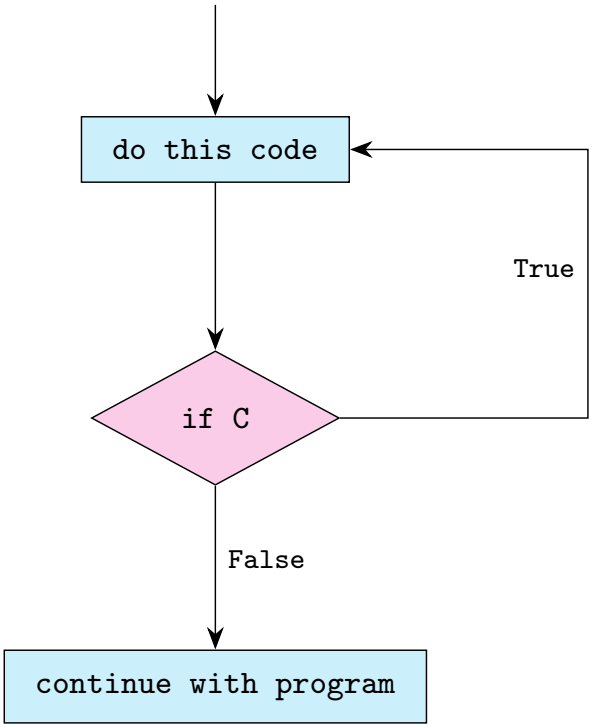

Figure 6: The do-while as a flowchart. Compare the while-loop figure in 2025-08-18-while-loops[28]: the body is above the test rather than below it, so the body always runs once before `C` is ever evaluated, and the `True` edge is the one that loops back.

Although not natively supported in Python, we can simulate one with `while True` and a `break` (some programmers prefer `1` to `True`):

```
>>> x = 0
>>> while 1:
...     x += 1
...     if x > 0:
...         break
>>> x            # x retains its value outside the while
1
```

`x` ends at `1`, not `0`: the body ran once before the exit test was reached, which is the whole point of the shape.

User input

`x = input()` waits for input from the keyboard and assigns it to `x`, always with `type(x) is str`. Numeric input therefore has to be converted — `int(input("Guess: "))`.

Random number generation

Random number generation is handled by an external library. It is **not** built-in and therefore must be imported. Importing the single command:

```
>>> from random import randint
>>> randint(1, 6)   # chosen uniformly over the interval
2
```

or the entire library:

```
>>> import random
>>> random.randint(1, 6)
2
```

Both endpoints are included: `randint(1, 6)` is the course's six-sided die, and its prompt reads `Guess in [1..6]`.

[28] courses/csse1001/2025-08-18-while-loops.pdf