

Day 11 Comprehensions and iteration tools

Self-Study Guide · About 75 minutes

By the end you will be able to

- Convert a list-building loop into a list comprehension, with or without a condition, and turn a comprehension back into a loop.
- Build dictionaries and sets with comprehensions, and use generator expressions with `sum()`, `any()`, and `all()`.
- Walk through several lists at once with `zip()`, number items with `enumerate()`, and go backwards with `reversed()`.
- Decide when a plain loop is clearer than a comprehension, and explain why.

What to remember from earlier

- An accumulator is set before a loop, updated inside it, and used after it. ([Day 5: Loops, 5.6 Counting and Summing](#))
- `for index, item in enumerate(sequence)` gives each item with its index, starting at 0 or at the number you pass. ([Day 5: Loops, 5.11 enumerate\(\): Index and Value Together](#))
- A conditional expression, such as "pass" if `score >= 60` else "fail", picks one of two values. ([Day 4: Making decisions, 4.9 Conditional Expressions](#))
- `split()` breaks a string into a list of words, and `separator.join(pieces)` glues pieces back into one string. ([Day 3: Strings, 3.7 Splitting and Joining](#))
- `append()` adds an item to the end of a list, and `sorted()` gives back a new sorted list without changing the original. ([Day 6: Lists](#))
- Unpacking puts each part of a tuple into its own name, as in `for name, score in pairs`. ([Day 7: Tuples, sets and unpacking, 7.3 Unpacking: Many Names at Once](#))
- A dictionary's `items()` gives each key and value as a pair, ready to unpack in a for loop. ([Day 8: Dictionaries, 8.5 Looping with keys\(\), values\(\), and items\(\)](#))

Make a `day11` folder for today's files, open it in your editor, and run each script from its terminal, as on earlier days.

Since Day 6 you have built many lists the same way: start with an empty list, loop over something, and append a result on each pass. Since Day 8 you have built dictionaries the same way too. The pattern works, but it takes three or four lines to say one thing: "a list of *this* for each item in *that*". Today you learn Python's one-line way to say it, the **comprehension**, plus a handful of built-in tools that make loops shorter and clearer: `zip`, `enumerate`, `reversed`, `sum`, `any`, and `all`. You will also learn the opposite skill, which matters just as much: recognizing when a plain loop is the better choice.

11.1 List Comprehensions

A weather app stores temperatures in Celsius and needs them in Fahrenheit. With what you know from Days 5 and 6, you would write the accumulator pattern for a list: an empty list before the loop, `append` inside it, and the finished list after it.

fahrenheit_loop.py

```

1 celsius = [0, 12, 25, 31]
2 fahrenheit = []
3 for c in celsius:
4     fahrenheit.append(c * 9 / 5 + 32)
5 print(fahrenheit)

```

```
[32.0, 53.6, 77.0, 87.8]
```

Look at what the four lines actually say: "fahrenheit is $c * 9 / 5 + 32$ for each c in celsius". Only two pieces of information matter: **what** to put in the new list ($c * 9 / 5 + 32$) and **where** the values come from (for c in celsius). The empty list and the append are scaffolding that every list-building loop repeats.

A **list comprehension** builds a new list by writing those two pieces inside square brackets. Here is the same program:

fahrenheit_comp.py

```

1 celsius = [0, 12, 25, 31]
2 fahrenheit = [c * 9 / 5 + 32 for c in celsius]
3 print(fahrenheit)

```

```
[32.0, 53.6, 77.0, 87.8]
```

The output is identical. Line 2 does the whole job of lines 2 to 4 in the loop version. Take it apart:

```
1 [expression for name in sequence]
```

- The **square brackets** say "the result is a new list", just as `[]` made an empty list on Day 6.
- The **expression** comes first: the value to put in the new list on each pass. Here, $c * 9 / 5 + 32$.
- **for name in sequence** is a for loop header without the colon. `name` is the loop variable, and it can be used in the expression.

Read it out loud left to right: "a list of $c * 9 / 5 + 32$ for each c in celsius". The expression is written first even though Python works it out on each pass, because the first thing a reader wants to know is what the list will hold.

WORKED EXAMPLE How Python runs a comprehension

For `[c * 9 / 5 + 32 for c in celsius]`, Python does exactly what the loop version did: it starts a new empty list, then for each item works out the expression and adds the result to the end.

Pass	c	$c * 9 / 5 + 32$	New list so far
1	0	32.0	[32.0]
2	12	53.6	[32.0, 53.6]
3	25	77.0	[32.0, 53.6, 77.0]
4	31	87.8	[32.0, 53.6, 77.0, 87.8]

After the last pass, the finished list is the value of the whole comprehension, and line 2 stores it in `fahrenheit`. The results are floats because `/` always gives a float (Day 2).

A comprehension can loop over anything a for loop can (any iterable, to use Day 10's word): a list, a string, a `range`, the words from `split()`. The expression can be any expression, including a method call or a function call:

```
>>> words = "to be or not".split()
>>> [word.upper() for word in words]
['TO', 'BE', 'OR', 'NOT']
>>> [len(word) for word in words]
[2, 2, 2, 3]
>>> [n * 10 for n in range(1, 5)]
[10, 20, 30, 40]
>>> [ch for ch in "hey"]
['h', 'e', 'y']
>>> [n * 10 for n in []]
[]
```

- `[word.upper() for word in words]` calls a string method on each word.
- `[len(word) for word in words]` calls a function on each word, giving a list of lengths.
- `[n * 10 for n in range(1, 5)]` loops over a range.
- `[ch for ch in "hey"]` uses the loop variable itself as the expression, so each character becomes an item.
- Looping over an empty list gives an empty list. There are no passes, so nothing is added; no error.

KEY IDEA

`[expression for name in sequence]` builds a **new** list holding the expression's value for each item. It does the same job as "empty list, for loop, append" in one line.

WATCH OUT

Expecting a comprehension to change the original list. A comprehension always builds a **new** list. It never changes the list it loops over, and if you do not store the new list, it is thrown away:

prices_unchanged.py

```
1 prices = [4, 7, 10]
2 [price * 2 for price in prices]
3 print(prices)
4 doubled = [price * 2 for price in prices]
5 print(doubled)
```

```
[4, 7, 10]
[8, 14, 20]
```

Line 2 builds `[8, 14, 20]`, but nothing names it, so it is lost the moment it is made, and `prices` is untouched. Line 4 stores the new list in `doubled`. If you want `prices` itself to hold the doubled values, assign the new list back to it: `prices = [price * 2 for price in prices]`.

PAUSE AND TRY

Question 1 Tracing

What does this program print?

word_sizes.py

```
1 animals = ["cat", "horse", "ox"]
2 sizes = [len(animal) - 1 for animal in animals]
3 print(sizes)
4 print(animals)
```

11.2 Filtering with if

Often you want only **some** of the items: the passing scores, the short words, the names that start with A. On Day 5 and Day 6 you did this with an `if` inside the loop, so only some passes called `append`:

passing_loop.py

```
1 scores = [72, 45, 90, 58, 61]
2 passing = []
3 for score in scores:
4     if score >= 60:
5         passing.append(score)
6 print(passing)
```

```
[72, 90, 61]
```

A comprehension can do the same. Add an `if` and a condition **after** the `for` part:

passing_comp.py

```
1 scores = [72, 45, 90, 58, 61]
2 passing = [score for score in scores if score >= 60]
3 print(passing)
```

```
[72, 90, 61]
```

```
1 [expression for name in sequence if condition]
```

For each item, Python first checks the condition. If it is truthy, the expression is worked out and added; if not, the item is skipped and nothing is added. The `if` part is a **filter**: it decides which items get through. Read line 2 as "a list of `score` for each `score` in `scores`, **if** `score >= 60`".

score	score >= 60	Added?
72	True	yes: 72
45	False	no
90	True	yes: 90
58	False	no
61	True	yes: 61

The filter and the expression can both do work. Here the names arrive with stray spaces. The condition uses `strip()` (Day 3) to measure the real name, and the expression cleans it up and makes it capitals:

long_names.py

```
1 names = [" ana ", "Bo", "carmen", "Li", "dmitri "]
2 long_names = [name.strip().upper() for name in names if len(name.strip()) > 2]
3 print(long_names)
```

```
['ANA', 'CARMEN', 'DMITRI']
```

"Bo" and "Li" are 2 characters after stripping, so the filter drops them. The other three pass the filter and the expression turns " ana " into "ANA". The condition is checked **first**, so the expression only runs for items that pass.

Filtering versus choosing

There is a second, different way an `if` can appear in a comprehension: as a **conditional expression** (Day 4), a `if condition else b`, which picks one of two values. That goes in the **expression** spot, at the front, and it does not drop anything: every item gives one value or the other.

filter_or_choose.py

```
1 scores = [72, 45, 90, 58, 61]
2 passing = [score for score in scores if score >= 60]
3 labels = ["pass" if score >= 60 else "fail" for score in scores]
4 print(passing)
5 print(labels)
```

```
[72, 90, 61]
['pass', 'fail', 'pass', 'fail', 'pass']
```

Where the <code>if</code> goes	What it is	Has <code>else</code> ?	Result length
--------------------------------	------------	-------------------------	---------------

After the <code>for</code> (at the end)	A filter	Never	Only the items that pass (3 here)
Before the <code>for</code> (at the front)	A conditional expression (Day 4)	Always	One value per item (5 here)

KEY IDEA

An `if` at the **end** of a comprehension filters: items that fail are left out. An `if ... else` at the **front** is a conditional expression: every item stays, and the condition chooses its value.

WATCH OUT

Putting `else` on the filter. It feels natural to finish "keep the score if it passes" with "else write fail":

filter_else_wrong.py

```
1 scores = [72, 45, 90, 58, 61]
2 labels = [score for score in scores if score >= 60 else "fail"]
3 print(labels)
```

```
File ".../day11/filter_else_wrong.py", line 2
  labels = [score for score in scores if score >= 60 else "fail"]
                                         ^^^^^
SyntaxError: invalid syntax
```

A filter has nothing to do with "else": a failed item is simply skipped, so Python does not allow an `else` there, and stops before running anything. If you want a value for every item, move the whole `if ... else` to the front as a conditional expression:

`[score if score >= 60 else "fail" for score in scores]`. Older versions of Python may word the message a little differently.

PAUSE AND TRY**Question 2** Tracing

What does this program print? Notice where each `if` is.

odd_or_zero.py

```
1 numbers = [3, 8, 5, 6]
2 print([n for n in numbers if n % 2 == 1])
3 print([n if n % 2 == 1 else 0 for n in numbers])
```

Question 3 Multiple choice

scores = [72, 45, 90, 38]. Which comprehension gives the scores below 50, each doubled, which is [90, 76]?

- A. `[score * 2 if score < 50 for score in scores]`
- B. `[score * 2 for score in scores if score < 50]`
- C. `[score for score in scores if score * 2 < 50]`
- D. `[score * 2 if score < 50 else score for score in scores]`

11.3 Reading a Comprehension as a Loop

Today's goal is to convert between a loop and a comprehension **in both directions**. Going from loop to comprehension lets you write shorter code; going from comprehension to loop lets you read code other people wrote, and debug it by adding `print` calls inside the loop.

The two are connected by a fixed recipe. Here is a loop and its comprehension, side by side:

short_words_loop.py

```
1 words = "the quick brown fox jumps over a lazy dog".split()
2 short = []
3 for word in words:
4     if len(word) <= 3:
5         short.append(word.upper())
6 print(short)
```

short_words_comp.py

```
1 words = "the quick brown fox jumps over a lazy dog".split()
2 short = [word.upper() for word in words if len(word) <= 3]
3 print(short)
```

Both print:

```
['THE', 'FOX', 'A', 'DOG']
```

Part	Loop version	Comprehension
New list	<code>short = []</code>	the square brackets <code>[]</code>
Where items come from	<code>for word in words:</code>	<code>for word in words</code>
Filter (optional)	<code>if len(word) <= 3:</code>	<code>if len(word) <= 3</code>
What gets added	<code>short.append(word.upper())</code>	<code>word.upper()</code> , moved to the front

Loop to comprehension. A loop can become a comprehension only if it has exactly this shape: an empty list, a `for`, at most an `if`, and a single `append`. Then:

1. Write the square brackets.
2. Copy what is inside `append(...)` to the front.
3. Copy the `for` line, without its colon.
4. If there is an `if`, copy it after the `for`, without its colon.
5. Assign the result to the list's name.

Comprehension to loop. Run the recipe backwards. Start with `name = []`. Turn everything after the expression into indented loop lines, in the same order, adding the colons back. Put `name.append(expression)` at the innermost level.

KEY IDEA

The clauses after the expression read like loop lines, **in the same order**, each one nested inside the one before. The expression is what the innermost line appends.

Two for clauses

That "same order" rule explains comprehensions with more than one `for`. Day 6 stored a grid as a list of lists. To get every number in one flat list, you need a nested loop (Day 5):

`flatten_loop.py`

```
1 grid = [[1, 2, 3], [4, 5, 6]]
2 flat = []
3 for row in grid:
4     for number in row:
5         flat.append(number)
6 print(flat)
```

The comprehension keeps the two `for` lines in the **same order**, outer first:

`flatten_comp.py`

```
1 grid = [[1, 2, 3], [4, 5, 6]]
2 flat = [number for row in grid for number in row]
3 print(flat)
```

Both print:

```
[1, 2, 3, 4, 5, 6]
```

Read `[number for row in grid for number in row]` as the nested loop: "for each `row` in `grid`, for each `number` in `row`, add `number`". The second `for` can use `row` because it is nested inside the first, just as the inner loop can.

A comprehension's expression can itself be a comprehension. That builds a list of lists, such as a small times table:

`times_grid.py`

```
1 grid = [[row * col for col in range(1, 4)] for row in range(1, 4)]
2 print(grid)
3 for line in grid:
4     print(line)
```

```
[[1, 2, 3], [2, 4, 6], [3, 6, 9]]
[1, 2, 3]
[2, 4, 6]
[3, 6, 9]
```

Work from the outside in. The outer comprehension makes one item for each `row` from 1 to 3. That item is the inner comprehension, `[row * col for col in range(1, 4)]`, which is a whole list: the

row's three products. So `grid` holds three lists. This is as far as a comprehension should go; any deeper and a loop is easier to read (11.10).

WATCH OUT

Writing the two `for` clauses in the wrong order. Thinking "a number for each number in each row", you might put the `for` clauses the other way round:

`flatten_wrong_order.py`

```
1 grid = [[1, 2, 3], [4, 5, 6]]
2 flat = [number for number in row for row in grid]
3 print(flat)
```

```
Traceback (most recent call last):
  File ".../day11/flatten_wrong_order.py", line 2, in <module>
    flat = [number for number in row for row in grid]
              ^^^
NameError: name 'row' is not defined. Did you mean: 'pow'?
```

Python reads the clauses in order, as nested loops, so the first `for` needs `row` before the second `for` has made it. (`pow` is a built-in function with a similar name; Python's guess is no help here.) Worse, if the program already had a `row` variable from earlier, there would be no error at all: the comprehension would quietly use the old `row` and give wrong results. Keep the clauses in the order the loops would be nested: outer first.

PAUSE AND TRY

Question 4 Short answer

Rewrite this loop as a list comprehension, and say what it prints.

`total_loop.py`

```
1 prices = [3.5, 12.0, 8.25, 20.0]
2 cheap = []
3 for price in prices:
4     if price < 10:
5         cheap.append(price * 2)
6 print(cheap)
```

11.4 Dictionary Comprehensions

On Day 8 you built dictionaries in a loop: start with `{}`, then `d[key] = value` on each pass. That is the same "empty, loop, add" scaffolding as a list, so Python has a comprehension for it too.

A **dictionary comprehension** builds a new dictionary. It uses curly braces, like a dictionary, and its expression is a key: value pair:

```
1 {key_expression: value_expression for name in sequence}
```

The colon between the two expressions is what makes it a dictionary. On each pass, Python works out the key and the value and adds that entry, exactly like `d[key] = value`.

WORKED EXAMPLE Each name and its length

name_lengths.py

```
1 names = ["Ana", "Bo", "Carmen"]
2 lengths = {name: len(name) for name in names}
3 print(lengths)
4 print(lengths["Carmen"])
```

```
{'Ana': 3, 'Bo': 2, 'Carmen': 6}
6
```

Pass	name	Key	Value	Dictionary so far
1	"Ana"	"Ana"	3	{'Ana': 3}
2	"Bo"	"Bo"	2	{'Ana': 3, 'Bo': 2}
3	"Carmen"	"Carmen"	6	{'Ana': 3, 'Bo': 2, 'Carmen': 6}

The result is an ordinary dictionary, so line 4 looks up a key with `[]` as on Day 8.

The most common source for a dictionary comprehension is another dictionary. `items()` (Day 8) gives each key and value as a pair, and unpacking (Day 7) puts them in two names. A café puts everything 50 cents off, and also wants just the items under 5:

menu_sale.py

```
1 menu = {"tea": 2.5, "cake": 4.0, "soup": 6.0}
2 sale = {item: price - 0.5 for item, price in menu.items()}
3 print(sale)
4 under_5 = {item: price for item, price in menu.items() if price < 5}
5 print(under_5)
```

```
{'tea': 2.0, 'cake': 3.5, 'soup': 5.5}
{'tea': 2.5, 'cake': 4.0}
```

Line 2 keeps every key and changes every value. Line 4 adds a filter at the end, exactly as in 11.2, so only items under 5 get an entry; the values are unchanged. `menu` itself is not touched by either line.

Because you choose both the key and the value, you can also swap them. An airport code table becomes a city-to-code table:

invert_codes.py

```
1 codes = {"SF0": "San Francisco", "JFK": "New York", "LHR": "London"}
2 cities = {city: code for code, city in codes.items()}
3 print(cities)
4 print(cities["London"])
```

```
{'San Francisco': 'SF0', 'New York': 'JFK', 'London': 'LHR'}
LHR
```

The loop unpacks each pair as `code`, `city`, and the expression puts `city` first, as the key.

KEY IDEA

`{key: value for name in sequence}` builds a new dictionary. With `for key, value in d.items()` you can change the values, filter the entries, or swap keys and values.

WATCH OUT

Two items that give the same key. A dictionary holds each key only once (Day 8). If two passes produce the same key, the later one replaces the earlier one, with no warning:

same_length_keys.py

```
1 words = ["cat", "dog", "horse"]
2 by_length = {len(word): word for word in words}
3 print(by_length)
```

```
{3: 'dog', 5: 'horse'}
```

"cat" and "dog" both have length 3. The pass for "cat" adds `3: 'cat'`, then the pass for "dog" overwrites it with `3: 'dog'`. Three words went in; two entries came out. The same happens when you swap keys and values and two keys shared a value. If several items can share a key and you need to keep them all, use the Day 8 grouping pattern with a loop, which collects a list for each key.

PAUSE AND TRY**Question 5** Tracing

What does this program print?

stock_check.py

```
1 stock = {"pens": 12, "pads": 0, "tape": 3, "glue": 0}
2 reorder = {item: 10 - count for item, count in stock.items() if count < 5}
3 print(reorder)
```

11.5 Set Comprehensions

On Day 7 you met sets: collections that keep only one copy of each value. To find the different words in a sentence, ignoring capitals, you want a set of lowercase words.

A **set comprehension** builds a new set. It uses curly braces like a dictionary comprehension, but its expression is a single value, with **no colon**:

```
1 {expression for name in sequence}
```

unique_words.py

```
1 text = "The cat saw the dog and the dog saw THE cat"
2 words = text.split()
3 unique = {word.lower() for word in words}
4 print(len(words), "words,", len(unique), "different")
5 print(sorted(unique))
```

```
11 words, 5 different
['and', 'cat', 'dog', 'saw', 'the']
```

`text.split()` gives 11 words. The set comprehension lowercases each one and adds it to the set; adding a value the set already holds changes nothing. So "The", "the", "the", and "THE" all end up as one "the". Five different words remain.

Why print `sorted(unique)` and not `unique`? A set has no order (Day 7), and the order Python shows a set of strings in can change from one run to the next. `sorted()` gives back a list in alphabetical order, which is the same every time.

Curly braces now do three jobs, and the difference is easy to miss:

```
>>> letters = {ch for ch in "banana"}
>>> sorted(letters)
['a', 'b', 'n']
>>> counts = {ch: "banana".count(ch) for ch in "banana"}
>>> counts
{'b': 1, 'a': 3, 'n': 2}
>>> type({})
<class 'dict'>
>>> type(set())
<class 'set'>
```

- `{ch for ch in "banana"}` has no colon, so it is a set: three different letters.
- `{ch: "banana".count(ch) for ch in "banana"}` has a colon, so it is a dictionary. (It counts each letter more than once on the way, which is wasteful, but the later entry for a repeated letter simply replaces the earlier one with the same count.)
- Empty `{}` is an empty **dictionary**, not an empty set. An empty set is written `set()`. `type()` shows which one you have.

KEY IDEA

`{expression for name in sequence}` builds a set: duplicates disappear. A colon inside the braces makes it a dictionary comprehension instead.

PAUSE AND TRY**Question 6** Tracing

What does this program print?

first_letters.py

```
1 names = ["Ana", "aLi", "Bo", "Carmen", "cLeo", "ben"]
2 firsts = {name[0].upper() for name in names}
3 print(sorted(firsts))
4 print(len(firsts))
```

11.6 Generator Expressions

Suppose you need the total of the squares of the numbers from 1 to a million. With a list comprehension, you can hand the list to `sum()` (Day 6):

big_sum.py

```
1 as_list = sum([n * n for n in range(1, 1000001)])
2 as_generator = sum(n * n for n in range(1, 1000001))
3 print(as_list)
4 print(as_generator)
```

```
333333833333500000
333333833333500000
```

Line 1 works, but think about what it does. The comprehension first builds a complete list with a **million** numbers in it, holding them all in the computer's memory at once. Only then does `sum()` start adding. Yet `sum()` never needs more than one number at a time: it adds the next number to its running total and moves on. The whole list is built just to be thrown away.

Line 2 uses **round brackets** instead of square ones, and gives the same answer without ever building the list. That is a **generator expression**: it looks like a list comprehension in parentheses, but instead of making all the values up front, it makes them **one at a time, when asked**. The object it gives back is called a **generator**. A generator is like a recipe card for a list rather than a tray of

cooked food: it holds the instructions and where it is up to, and produces the next value only when something asks for it.

```
1 (expression for name in sequence if condition)
```

Everything you learned about the parts still applies, including the filter. Only the brackets change.

Watching a generator be lazy

You cannot see memory directly, but you can see **when** values are made. This program uses a function (Day 9) that prints a message every time it is called, so you can watch:

WORKED EXAMPLE Made when asked

lazy_squares.py

```
1 def square(n):
2     print("making", n)
3     return n * n
4
5
6 squares = (square(n) for n in range(1, 4))
7 print("generator ready")
8 for value in squares:
9     print("got", value)
```

```
generator ready
making 1
got 1
making 2
got 4
making 3
got 9
```

Line 6 creates the generator, but `square` is not called at all: `generator ready` prints first. Then the for loop asks for values one by one. Each time, the generator calls `square` for the next number and hands over the result, and the loop body prints it before the next value is made. At any moment, only one square exists.

Now change only the brackets to square ones:

eager_squares.py

```
1 def square(n):
2     print("making", n)
3     return n * n
4
5
6 squares = [square(n) for n in range(1, 4)]
7 print("list ready")
8 for value in squares:
9     print("got", value)
```

```
making 1
making 2
making 3
list ready
got 1
got 4
got 9
```

The list comprehension makes all three values **before** line 7 runs. That is the whole difference: a list comprehension does all the work at once and keeps every result; a generator expression does the work as the values are used and keeps none of them. With three values it does not matter; with a million, the list version holds a million numbers in memory and the generator version holds one.

Printing a generator does not show its values. Python shows a short description that starts `<generator object` and ends with a memory address, which changes from run to run. To see the values, turn it into a list with `list()`:

```
>>> doubled = (n * 2 for n in [5, 6, 7])
>>> list(doubled)
[10, 12, 14]
>>> list(doubled)
[]
```

The second `list(doubled)` is empty. That is not a mistake in the transcript; it is the next rule.

A generator can be used only once

A generator remembers where it is up to, and once it has handed over its last value, it is **used up**: asking again gives nothing.

used_up.py

```
1 squares = (n * n for n in range(1, 4))
2 print(sum(squares))
3 print(sum(squares))
4 print(list(squares))
```

```
14
0
[]
```

The first `sum` uses all three values ($1 + 4 + 9 = 14$). The second `sum` finds nothing left, and the sum of no numbers is 0. `list(squares)` gives an empty list. If you need the values more than once, use a list comprehension and keep the list.

Feature	List comprehension [...]	Generator expression (...)
When values are made	All at once, immediately	One at a time, when asked
Memory	Holds every value	Holds one value at a time
Use it more than once?	Yes	No, it is used up
<code>len()</code> and indexing	Yes	No: it does not know its length

Best for

Keeping a collection

Feeding a function such as `sum`**KEY IDEA**

A generator expression (`expression for name in sequence`) makes its values one at a time, as they are used, so it never holds them all in memory. It can be used only once.

When a generator expression is the **only** argument to a function, you can drop its own parentheses: `sum(n * n for n in numbers)` is `sum((n * n for n in numbers))` with one pair of brackets doing both jobs. Round brackets with a `for` inside never make a tuple; Python has no tuple comprehension.

WATCH OUT

Leaving out the parentheses when there is another argument. `sorted()` can take a generator expression, and `reverse=True` (Day 6) as well:

two_args_wrong.py

```
1 names = ["bo", "Ana", "carmen"]
2 ordered = sorted(name.lower() for name in names, reverse=True)
3 print(ordered)
```

```
File ".../day11/two_args_wrong.py", line 2
    ordered = sorted(name.lower() for name in names, reverse=True)
                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

SyntaxError: Generator expression must be parenthesized

With a second argument, Python cannot tell where the generator expression ends, so it asks for brackets around it. Give it its own pair:

two_args_fixed.py

```
1 names = ["bo", "Ana", "carmen"]
2 ordered = sorted((name.lower() for name in names), reverse=True)
3 print(ordered)
```

```
['carmen', 'bo', 'ana']
```

Older versions of Python may word the message differently.

PAUSE AND TRY**Question 7** Tracing

What does this program print? Think about what is left in `evens` after each line.

evens_twice.py

```
1 evens = (n for n in range(10) if n % 2 == 0)
2 print(max(evens))
3 print(len([n for n in range(10) if n % 2 == 0]))
4 print(sum(evens))
```

Question 8 Multiple choice

You need to know the total length of every word in a 50 000-word book, once. Which is the best choice?

- A. `sum([len(word) for word in words])`, because a list is faster to make
- B. `sum(len(word) for word in words)`, because no list of 50 000 lengths has to be built
- C. `(len(word) for word in words)`, because a generator is already a total
- D. `len(word for word in words)`, because `len` adds up lengths

11.7 `sum()`, `any()` and `all()`

Generator expressions are most useful as the argument to a function that walks through values and boils them down to one answer. You know `sum()`, `min()`, and `max()` from Day 6. Two more built-in functions answer yes-or-no questions about a whole collection:

- `any(values)` gives `True` if **at least one** value is truthy, and `False` otherwise.
- `all(values)` gives `True` if **every** value is truthy, and `False` otherwise.

Combined with a generator expression, each of these replaces a whole accumulator loop from Day 5:

`sum_any_all.py`

```
1 scores = [72, 45, 90, 58, 61]
2 print(sum(1 for score in scores if score >= 60))
3 print(max(score - 50 for score in scores))
4 print(any(score >= 90 for score in scores))
5 print(all(score >= 50 for score in scores))
```

```
3
40
True
False
```

Line	Question it answers	How	Answer
2	How many scores are 60 or more?	Add 1 for each score that passes the filter	3
3	How far is the best score above 50?	Largest of <code>score - 50</code>	40
4	Did anyone score 90 or more?	Is at least one <code>score >= 90</code> true?	True
5	Did everyone score at least 50?	Is every <code>score >= 50</code> true? 45 is not	False

Line 2 is the **counting** pattern in one line: `sum(1 for ... if ...)` adds a 1 for every item that passes the filter, which is exactly what a Day 5 counter did with `count += 1`. Here is the Day 5 vowel counter rewritten that way:

`count_vowels_gen.py`

```
1 text = "Programming is fun"
2 vowels = sum(1 for ch in text.lower() if ch in "aeiou")
3 print("Vowels:", vowels)
```

Vowels: 5

Lines 4 and 5 have a condition as the expression, so the generator gives `True` or `False` for each score, and `any` or `all` checks those.

`any()` and `all()` stop early

`any` and `all` work like `or` and `and` (Day 4): they stop as soon as the answer is decided. `any` stops at the first truthy value, because one is enough; `all` stops at the first falsy value, because one failure is enough. Combined with a generator, that means the remaining values are never even made:

`any_stops_early.py`

```
1 def too_cold(temp):
2     print("checking", temp)
3     return temp < 0
4
5
6 temps = [4, -2, 7, 1]
7 print(any(too_cold(t) for t in temps))
8 print(all(t > 0 for t in temps))
```

```
checking 4
checking -2
True
False
```

`too_cold` is called for 4 (not below 0), then for -2 (below 0). That is enough for `any`, so 7 and 1 are never checked. If you wrote `any([too_cold(t) for t in temps])` with square brackets, the list comprehension would call `too_cold` for all four temperatures first; this is one more reason to pass a generator expression rather than a list.

Empty collections

What should "did everyone pass?" say about a class with no students? Python's answer is `True`, because there is no student who failed. `any` of nothing is `False`, because there is no value that is true:

```
>>> any([])
False
>>> all([])
True
>>> scores = []
>>> all(score >= 60 for score in scores)
True
>>> len(scores) > 0 and all(score >= 60 for score in scores)
False
```

This is logically right but can surprise you. If an empty list should count as "not passed", check that the list has items first, as the last line does. The `and` stops at `len(scores) > 0`, which is `False`.

KEY IDEA

`any(condition for item in items)` asks "is it true for at least one?"; `all(...)` asks "is it true for every one?". Both stop as soon as they know. `all` of an empty collection is `True`, and `any` of one is `False`.

WATCH OUT

`all` without a condition. `all(scores)` asks whether every score is **truthy**, not whether every score passes. Any number other than 0 is truthy (Day 4), so `all([72, 45, 3])` is `True`. Always write the condition you mean: `all(score >= 60 for score in scores)`.

PAUSE AND TRY**Question 9** Tracing

What does this program print?

password_rules.py

```
1 password = "Sunny5day"
2 has_digit = any(ch in "0123456789" for ch in password)
3 all_lower = all(ch == ch.lower() for ch in password)
4 long_enough = len(password) >= 8
5 print(has_digit, all_lower, long_enough)
```

11.8 zip(): Walking Lists Side by Side

Often two lists belong together: `names[0]` goes with `scores[0]`, `names[1]` with `scores[1]`, and so on. Lists that line up position by position like this are called **parallel lists**. With what you knew before today, you would loop over the indexes:

index_pairs.py

```
1 names = ["Ana", "Bo", "Carmen"]
2 scores = [88, 92, 79]
3 for i in range(len(names)):
4     print(names[i], scores[i])
```

```
Ana 88
Bo 92
Carmen 79
```

It works, but the index `i` is only a means to an end, and every use of it is a chance to write the wrong list or the wrong index. You want to go through **both lists together**.

The built-in function `zip()` takes two or more sequences and pairs up their items by position: the first item of each, then the second of each, and so on. Each pair is a tuple, so a for loop can unpack it (Day 7) into two names:

zip_pairs.py

```
1 names = ["Ana", "Bo", "Carmen"]
2 scores = [88, 92, 79]
3 for name, score in zip(names, scores):
4     print(name, score)
5 print(list(zip(names, scores)))
```

```
Ana 88
Bo 92
Carmen 79
[('Ana', 88), ('Bo', 92), ('Carmen', 79)]
```

Think of a zipper: two rows of teeth, joined one pair at a time. Line 3 unpacks each tuple into `name` and `score`. Line 5 uses `list()`, which builds a list from the pairs, so you can see what `zip` makes. Like a generator, `zip` makes its pairs one at a time as they are asked for; printing a `zip` directly shows only a description, not the pairs.

`zip` takes as many sequences as you like. With three, each item is a tuple of three, and you unpack it into three names. Because `zip` gives pairs one at a time, it works inside a generator expression too:

zip_three.py

```
1 items = ["pen", "pad", "tape"]
2 prices = [1.5, 3.0, 2.25]
3 counts = [4, 2, 1]
4 for item, price, count in zip(items, prices, counts):
5     print(f"{count} x {item}: {price * count:.2f}")
6 total = sum(price * count for price, count in zip(prices, counts))
7 print(f"Total: {total:.2f}")
```

```
4 x pen: 6.00
2 x pad: 6.00
1 x tape: 2.25
Total: 14.25
```

Line 6 multiplies each price by its count and sums the results, with no index anywhere.

Two parallel lists of keys and values are exactly what a dictionary needs. A dictionary comprehension can build one, and so can `dict()`, which builds a dictionary directly from a sequence of key-value pairs:

zip_dict.py

```
1 names = ["Ana", "Bo", "Carmen"]
2 scores = [88, 92, 79]
3 grades = {name: score for name, score in zip(names, scores)}
4 print(grades)
5 print(dict(zip(names, scores)))
```

```
{'Ana': 88, 'Bo': 92, 'Carmen': 79}
```

```
{'Ana': 88, 'Bo': 92, 'Carmen': 79}
```

Comparing neighbors

zip can also pair a list with a **shifted copy of itself**. temps[1:] (a Day 6 slice) is the list without its first item, so zipping temps with temps[1:] pairs each value with the next one. Then all can check a rule for every neighboring pair:

rising.py

```
1 temps = [11, 13, 13, 16]
2 pairs = list(zip(temps, temps[1:]))
3 print(pairs)
4 print(all(later > earlier for earlier, later in pairs))
5 print(all(later >= earlier for earlier, later in pairs))
```

```
[(11, 13), (13, 13), (13, 16)]
```

```
False
```

```
True
```

Four temperatures give three neighboring pairs. "Each day strictly warmer" fails at (13, 13); "never colder" holds for all three.

KEY IDEA

zip(a, b) pairs the items of a and b by position, as tuples. for x, y in zip(a, b): walks both lists at once with no index.

WATCH OUT

Lists of different lengths. zip stops as soon as the **shortest** sequence runs out, and it does not warn you:

zip_short.py

```
1 names = ["Ana", "Bo", "Carmen", "Dee"]
2 scores = [88, 92, 79]
3 for name, score in zip(names, scores):
4     print(name, score)
5 print(len(names), len(scores))
```

```
Ana 88
Bo 92
Carmen 79
4 3
```

Dee has no score, so she is silently left out. That is useful when you mean it, as with `temps[1:]` above, but when the lists are supposed to match it hides a bug. If you are not sure, compare `len()` of the lists first.

PAUSE AND TRY

Question 10 Tracing

What does this program print?

zip_trace.py

```
1 first = ["a", "b", "c", "d"]
2 second = [1, 2, 3]
3 print([letter * number for letter, number in zip(first, second)])
```

11.9 enumerate() and reversed()

On Day 5 you used `enumerate()` on strings: `for index, item in enumerate(sequence):` gives each item with its index, from 0, or from the number you pass as a second argument. Now that you have lists and unpacking, you can see what it really does, and use it in more places.

`enumerate` works on a list exactly as on a string. And like `zip`, it makes tuples, one `(index, item)` pair per item; that is why the two-name `for` line works, and why Day 5's single-name loop printed pairs in parentheses:

numbered_tasks.py

```
1 tasks = ["buy milk", "call Bo", "water plants"]
2 for number, task in enumerate(tasks, 1):
3     print(f"{number}. {task}")
4 print(list(enumerate(tasks, 1)))
```

```
1. buy milk
2. call Bo
3. water plants
[(1, 'buy milk'), (2, 'call Bo'), (3, 'water plants')]
```

Because it gives pairs one at a time, `enumerate` also works inside a comprehension. Unpack the pair in the `for` part, and use either name in the expression or the filter:

numbered_comp.py

```
1 tasks = ["buy milk", "call Bo", "water plants"]
2 lines = [f"{number}. {task}" for number, task in enumerate(tasks, 1)]
3 print(lines)
4 every_other = [task for index, task in enumerate(tasks) if index % 2 == 0]
5 print(every_other)
```

```
['1. buy milk', '2. call Bo', '3. water plants']
['buy milk', 'water plants']
```

Line 2 builds the numbered lines without printing them, ready to use later. Line 4 filters on the **index**: `index % 2 == 0` keeps positions 0 and 2, which is every other task.

Going backwards

To go through a list from the end, you have seen two tools on Day 6: the `reverse()` method and the slice `[::-1]`. A third, the built-in function `reversed()`, gives the items of a sequence from last to first, one at a time, **without changing or copying** the sequence:

backwards.py

```
1 laps = [62, 60, 59]
2 for lap in reversed(laps):
3     print(lap)
4 print(laps)
5 print(list(reversed(laps)))
6 print(list(reversed("stop")))
7 print(list(reversed(range(1, 4))))
```

```
59
60
62
[62, 60, 59]
[59, 60, 62]
['p', 'o', 't', 's']
[3, 2, 1]
```

The loop visits the laps from last to first, and printing `laps` afterwards shows it is unchanged.

`reversed` works on a string or a `range` too. Like `zip`, it hands its items over one at a time, so wrap it in `list()` to see them all at once.

All three ways of going backwards side by side:

three_reverses.py

```

1 laps = [62, 60, 59]
2 copy = laps[::-1]
3 print(copy, laps)
4 backwards = list(reversed(laps))
5 print(backwards, laps)
6 result = laps.reverse()
7 print(result, laps)

```

```

[59, 60, 62] [62, 60, 59]
[59, 60, 62] [62, 60, 59]
None [59, 60, 62]

```

Tool	Changes laps?	Gives back	Use when
<code>laps[::-1]</code>	No	A new, reversed list (a copy)	You want to keep a reversed list
<code>reversed(laps)</code>	No	Items one at a time, last first	You only want to loop backwards
<code>laps.reverse()</code>	Yes, in place	None	You want the list itself turned around

KEY IDEA

`enumerate()` and `reversed()` give their items one at a time and work in loops and comprehensions alike. `reversed()` goes backwards without changing the list; `list.reverse()` changes the list and returns `None`.

WATCH OUT

Looping over `list.reverse()`. Since `reverse()` returns `None`, using it as the thing to loop over fails:

reverse_none.py

```

1 laps = [62, 60, 59]
2 for lap in laps.reverse():
3     print(lap)

```

```

Traceback (most recent call last):
  File ".../day11/reverse_none.py", line 2, in <module>
    for lap in laps.reverse():
                   ~~~~~^
TypeError: 'NoneType' object is not iterable

```

An iterable, as Day 10 defined it, is anything a for loop can go through, and `None` is not one. (The list did get reversed before the error.) Write `for lap in reversed(laps):` instead. Older versions of Python may show the error line differently.

PAUSE AND TRY**Question 11** Tracing

What does this program print?

countdown_enumerate.py

```
1 steps = ["mix", "bake", "cool"]
2 for number, step in enumerate(reversed(steps), 1):
3     print(number, step)
```

11.10 When a Plain Loop Is Clearer

Comprehensions are shorter, but shorter is not the goal; **clear** is. Code is read far more often than it is written, by other people and by you a month from now. A comprehension is clearer than a loop only when the reader can take in the whole line at once.

A comprehension is a good fit when all of these are true:

- The goal is to **build a new collection** (or feed `sum`, `any`, `all`, `min`, or `max`).
- Each item gives **one value** from **one expression**.
- There is at most **one simple filter**, and at most two `for` clauses.

Use a plain loop when any of these is true:

Situation	Why a loop is better
The point is an action, such as <code>print</code>	A comprehension builds a list you do not want
Several cases with <code>if</code> / <code>elif</code> / <code>else</code>	Chained conditional expressions are hard to read
Several steps per item, or several variables	A comprehension has room for one expression
You need to stop early with <code>break</code>	A comprehension cannot <code>break</code>
You need to see what is happening	You can put a <code>print</code> inside a loop to debug it

Here is the first case. It is tempting to "save lines" by writing a comprehension whose expression is a `print`:

WATCH OUT

A comprehension used only for its side effect.

print_comp.py

```
1 names = ["Ana", "Bo"]
2 result = [print("Hello,", name) for name in names]
3 print(result)
```

```
Hello, Ana
Hello, Bo
[None, None]
```

The greetings do print, but `print`, like any function with no value to give back, returns `None` (Day 9), so the comprehension also builds a list of `None`s that nobody needs. A reader sees square brackets and expects a list that matters. When the point is an action, write the loop:

print_loop.py

```
1 names = ["Ana", "Bo"]
2 for name in names:
3     print("Hello,", name)
```

```
Hello, Ana
Hello, Bo
```

The second case: turning scores into letter grades has four outcomes. Conditional expressions can be chained, so a comprehension **can** do it:

grades_comp.py

```
1 scores = [95, 82, 71, 64]
2 grades = [
3     "A" if s >= 90 else "B" if s >= 80 else "C" if s >= 70 else "F"
4     for s in scores
5 ]
6 print(grades)
```

```
['A', 'B', 'C', 'F']
```

It works, but to check it you have to untangle "A" if ... else "B" if ... else ... in your head. The Day 4 `if / elif` chain says the same thing one rule per line:

grades_loop.py

```
1 scores = [95, 82, 71, 64]
2 grades = []
3 for score in scores:
4     if score >= 90:
5         grades.append("A")
6     elif score >= 80:
7         grades.append("B")
8     elif score >= 70:
9         grades.append("C")
10    else:
11        grades.append("F")
12 print(grades)
```

```
['A', 'B', 'C', 'F']
```

Same result, longer code, much easier to read and to change. An even better option, from Day 9, is to put the `if / elif` chain in a function such as `letter_grade(score)`; then the comprehension `[letter_grade(score) for score in scores]` is short **and** clear, because the hard part has a name.

KEY IDEA

Use a comprehension to build a collection from one expression and at most one simple filter. Use a loop for actions like printing, for several cases or steps, for `break`, and whenever the comprehension would not fit easily in one reader's glance.

PAUSE AND TRY**Question 12** Short answer

For each job, say whether you would write a comprehension or a plain loop, and why: (a) a list of every price with 20% added; (b) print a receipt line for each item; (c) find the first negative number in a long list and stop there; (d) a set of the different cities in a list of addresses.

Putting it together

A teacher has two quiz scores (out of 10) for five students, kept in three parallel lists. The program reports each student's total, the class average, who improved, whether anyone was perfect on quiz 2, whether everyone reached 12, and the top three.

quiz_report.py

```

1 names = ["Ana", "Bo", "Carmen", "Dee", "Eli"]
2 quiz1 = [8, 6, 9, 4, 7]
3 quiz2 = [9, 5, 10, 6, 7]
4
5 totals = {name: a + b for name, a, b in zip(names, quiz1, quiz2)}
6 print(totals)
7
8 average = sum(totals.values()) / len(totals)
9 print(f"Average: {average:.1f}")
10
11 improved = [name for name, a, b in zip(names, quiz1, quiz2) if b > a]
12 print("Improved:", improved)
13
14 print("Anyone perfect on quiz 2?", any(score == 10 for score in quiz2))
15 print("Everyone passed?", all(total >= 12 for total in totals.values()))
16
17 ranking = sorted(totals, key=lambda name: totals[name], reverse=True)
18 for place, name in enumerate(ranking[:3], 1):
19     print(f"{place}. {name} ({totals[name]})")

```

```

{'Ana': 17, 'Bo': 11, 'Carmen': 19, 'Dee': 10, 'Eli': 14}
Average: 14.2
Improved: ['Ana', 'Carmen', 'Dee']
Anyone perfect on quiz 2? True
Everyone passed? False
1. Carmen (19)
2. Ana (17)
3. Eli (14)

```

Line	What it does	Idea
5	<code>zip</code> walks three lists at once; the dictionary comprehension maps <code>name</code> → <code>total</code>	11.8, 11.4
8	<code>sum</code> of the dictionary's values, 71, divided by 5 entries, is 14.2	11.7
11	List comprehension with a filter: keep the names whose second score is higher	11.2
14	<code>any</code> stops at Carmen's 10	11.7
15	<code>all</code> fails at Bo's 11 and stops	11.7
17	<code>sorted</code> with a <code>lambda</code> key (Day 10) orders the names by total, highest first	Day 10
18	<code>enumerate(..., 1)</code> numbers the first three names of the ranking	11.9

- Every collection is built in one line, but the final report uses a **loop**, because printing is an action (11.10).
- Line 11 zips the lists again rather than using `totals`, because it needs both quiz scores, not their sum.
- Line 17 loops over `totals`, which gives its keys (Day 8), so `ranking` is a list of names; the `lambda` looks up each name's total to sort by.

Common mistakes

Mistake	What happens	Fix
<code>[x for x in items if cond else y]</code>	<code>SyntaxError: invalid syntax</code>	Move <code>y if cond else ...</code> to the front, or drop the <code>else</code>
Expecting a comprehension to change the original list	The original is unchanged; an unsaved result is lost	Assign the result: <code>items = [... for ... in items]</code>
Two items give the same key in a dictionary comprehension	The later one replaces the earlier one silently	Use a loop that groups values into lists (Day 8)
Using a generator twice	The second use finds nothing: <code>sum</code> gives 0, <code>list</code> gives <code>[]</code>	Use a list comprehension if you need the values again
<code>len()</code> of a generator	<code>TypeError: object of type 'generator' has no len()</code>	Build a list instead
<code>sorted(x for x in items, reverse=True)</code>	<code>SyntaxError: Generator expression must be parenthesized</code>	<code>sorted((x for x in items), reverse=True)</code>
<code>zip</code> on lists of different lengths	Extra items are silently dropped	Check the lengths first
<code>for x in items.reverse():</code>	<code>TypeError: 'NoneType' object is not iterable</code>	<code>for x in reversed(items):</code>
A comprehension used just to print	Prints, and builds a useless list of <code>None</code>	Use a <code>for</code> loop

Vocabulary

Term	Definition	Example
comprehension	A one-line expression that builds a collection from a loop	<code>[n * 2 for n in nums]</code>
list comprehension	Builds a new list: an expression for each item, in square brackets	<code>[len(w) for w in words]</code>
filter	An <code>if</code> at the end of a comprehension that decides which items are included	<code>[s for s in scores if s >= 60]</code>
dictionary comprehension	Builds a new dictionary from <code>key: value</code> expressions, in curly braces	<code>{w: len(w) for w in words}</code>
set comprehension	Builds a new set from one expression, in curly braces, with no colon	<code>{w.lower() for w in words}</code>
generator expression	Like a list comprehension in round brackets, but makes its values one at a time	<code>(n * n for n in nums)</code>
generator	The object a generator expression gives; it hands out values when asked, only once	<code>squares = (n * n for n in nums)</code>

<code>any()</code>	True if at least one value is truthy; stops at the first one	<code>any(s >= 90 for s in scores)</code>
<code>all()</code>	True if every value is truthy; stops at the first falsy one	<code>all(s >= 50 for s in scores)</code>
parallel lists	Lists whose items belong together by position	<code>names and scores</code>
<code>zip()</code>	Pairs the items of several sequences by position, as tuples	<code>zip(names, scores)</code>
<code>reversed()</code>	Gives a sequence's items last to first, without changing it	<code>reversed(laps)</code>

Summary

- A **list comprehension**, `[expression for name in sequence if condition]`, builds a new list in one line; the optional `if` at the end is a **filter**. An `if ... else` at the front is a conditional expression that keeps every item.
- A comprehension and a loop convert by a fixed recipe: the clauses after the expression are the loop lines in the same order, and the expression is what gets appended.
- **Dictionary comprehensions** `{key: value for ...}` and **set comprehensions** `{value for ...}` use curly braces; the colon makes the difference.
- A **generator expression** `(... for ...)` makes values one at a time, so it saves memory, but it can be used only once. It is the natural argument for `sum`, `min`, `max`, `any`, and `all`, and `any` and `all` stop as soon as the answer is known.
- `zip` walks parallel lists together, `enumerate` numbers items, and `reversed` goes backwards without changing the list. Use a plain loop when the work is an action, has several cases or steps, or needs `break`.

Practice

Do these in your `day11` folder. Type each program yourself, run it, and only then open the answer.

Question 13 Short answer

Rewrite three earlier loops as comprehensions. Choose three loops you wrote on earlier days that build a list, a string, or a count, and rewrite each with a comprehension or generator expression. Print the loop's result and the comprehension's result side by side to check that they match.

Question 14 Short answer

Word lengths. Build a dictionary that maps each word in a sentence to its length.

Question 15 Short answer

Pairs and a rule. Use `zip` to pair two lists, then use `all` to check that every pair satisfies a rule. For example: planned minutes and actual minutes for four tasks, and the rule "no task took longer than planned".

Review questions

Question 16 Tracing

What does this program print?

stock_trace.py

```
1 items = ["pens", "pads", "tape", "glue"]
2 counts = [12, 0, 3, 7]
3 low = {item: count for item, count in zip(items, counts) if count < 8}
4 print(low)
5 print(any(count == 0 for count in low.values()))
```

Question 17 Debugging

This program should print the average of the scores in a comma-separated string. It stops with an error instead. Explain the error and fix the program.

average_gen.py

```
1 text = "72,90,61"
2 scores = (int(part) for part in text.split(","))
3 print(sum(scores) / len(scores))
```

Question 18 Short answer

Rewrite this comprehension as nested for loops that build the same list.

pairs_comp.py

```
1 pairs = [(a, b) for a in range(1, 4) for b in range(1, 4) if a < b]
2 print(pairs)
```

Question 19 Tracing

A runner records lap times in seconds. What does this program print?

last_laps.py

```
1 laps = [62, 60, 61, 58]
2 faster = [later < earlier for earlier, later in zip(laps, laps[1:])]
3 print(faster)
4 for number, lap in enumerate(reversed(laps), 1):
5     if number > 2:
6         break
7     print(number, lap)
```

Question 20 Multiple choice

What does this program print?

colors_count.py

```
1 colors = ["red", "Blue", "red", "green", "BLUE", "red"]
2 as_list = [c.lower() for c in colors]
3 as_set = {c.lower() for c in colors}
4 print(len(as_list), len(as_set))
```

- A. 6 6
- B. 6 3
- C. 3 3
- D. 6 4

Question 21 Short answer

A classmate wrote this comprehension. Say what it prints, then rewrite it as a plain loop and explain which version you would rather read.

tangled.py

```
1 words = ["apple", "kiwi", "banana", "fig", "cherry"]
2 picked = [w.upper() if len(w) > 5 else w for w in words if "a" in w or "i" in w]
3 print(picked)
```

Go further

The official Python Tutorial covers today's topics in Data Structures: List Comprehensions, Nested List Comprehensions, and Looping Techniques, which shows `zip`, `enumerate`, and `reversed` together.

Reflection

Check each statement. If the answer is "not yet," reread the linked subsection.

☐ I can write a list comprehension that builds a new list from another. Reread: 11.1 List Comprehensions

☐ I can filter with an `if` at the end and tell it apart from a conditional expression at the front.

Reread: 11.2 Filtering with `if`

☐ I can turn a loop into a comprehension and a comprehension back into a loop. Reread: 11.3

Reading a Comprehension as a Loop

☐ I can build a dictionary with a dictionary comprehension. Reread: 11.4 Dictionary Comprehensions

☐ I can build a set with a set comprehension and tell it apart from a dictionary. Reread: 11.5 Set Comprehensions

☐ I can explain why a generator expression saves memory and why it can be used only once.

Reread: 11.6 Generator Expressions

☐ I can count, total, and check rules with `sum()`, `any()`, and `all()`. Reread: 11.7 `sum()`, `any()` and `all()`

☐ I can walk parallel lists together with `zip()`. Reread: 11.8 `zip()`: Walking Lists Side by Side

☐ I can number items with `enumerate()` and go backwards with `reversed()`. Reread: 11.9 `enumerate()` and `reversed()`

☐ I can decide when a plain loop is clearer than a comprehension. Reread: 11.10 When a Plain Loop Is Clearer