PYTHON DROPS

Short, actionable Python tips

Python drops

Rodrigo Girão Serrão

Jan 29, 2026

Contents

What is this book?

This book collects the daily Python tips that I send to the mathspp drops newsletter. Short and actionable tips to make you smarter about Python.

How to read this book?

Browse it. Open it in a random page. Pick a random number and read the tip corresponding to that number. Do whatever you want, the book is yours!

1 – `zip`'s keyword argument `strict`

The Python built-in `zip` has a keyword argument `strict` that will raise an error if the 2 (or more) iterables that you pass to `zip` don't have the same length.

Use this whenever you are passing arguments that should have the same length: it helps catch errors early.

Beware that `zip` only raises the error when it reaches the end of the shortest iterable. In other words, it doesn't validate the lengths upfront.

That's why you are able to print the first two names, and only then `zip` raises a `ValueError` when the list `lasts` ends:

```
firsts = ["Luke", "Darth", "Obi-Wan"]
lasts = ["Skywalker", "Vader"]
for first_name, last_name in zip(firsts, lasts, strict=True):
    print(f"{first_name} {last_name}")
```

```
Luke Skywalker
Darth Vader
Traceback (most recent call last):
  File "<python-input-0>", line 3, in <module>
    for first_name, last_name in zip(firsts, lasts, strict=True):
                                 ~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ValueError: zip() argument 2 is shorter than argument 1
```

Further reading:

- Article about `zip`

2 – Case-insensitive string comparisons

To perform case-insensitive string comparisons in Python, use the method `str.casefold`. This method exists precisely to let you perform case-insensitive comparisons.

You need it because some characters in some languages are kinda funky! (That's the technical term – funky.)

Using `lower` or `upper` for case-insensitive comparisons only works if you're working with strings that are 100% guaranteed to only contain ASCII characters. If you are working with Unicode, you need `casefold`.

Here is a small example using the German word for "street". First, note how the string "straße" appears to be lowercase, but if you uppercase it and then lowercase it, you end up with a different string:

```
print("straße".lower())
# straße

print("straße".upper().lower())
# strasse
```

Now, note how the method `casefold` works just fine:

```
print("STRASSE".casefold() == "straße".casefold())
# True

print("straße".casefold())
# strasse
```

Further reading:

- How to work with case-insensitive strings

3 – Type unions with the vertical bar in `isinstance`

The vertical bar `|` can be used to combine types (to create type unions) since Python 3.10. This lets you create type unions in a very ergonomic way.

You can use it, for example, inside `isinstance` checks. This lets you check if a value belongs to one of two or more types, like so:

```
isinstance(x, typ1 | typ2 | ...)
```

This is definitely nicer than

```
isinstance(x, typ1) or isinstance(x, typ2) or ...
```

And it is also nicer than the traditional alternative with a tuple of types:

```
isinstance(x, (typ1, typ2, ...))
```

Here is a complete example:

```
def is_number(x):
    return isinstance(x, int | float | complex)

print(is_number(42))  # True
print(is_number("hey"))  # False
```

4 – Parsing integers from different bases

The built-in `int` can be used to parse integers from binary, octal, hexadecimal, and many other bases. To do this, you need to specify the base you want as the second argument of `int`.

The valid bases are 2 through 36. (You use 2 for binary, 8 for octal, and 16 for hexadecimal.)

```
print(int("13"))  # 13
print(int("101", 2))  # 5
print(int("ff", 16))  # 255
```

The second argument can also be the special value 0, which tells `int` to "guess" the base by parsing the integer as it if were an integer literal. (It "guesses" the base if you use the base prefix: `0b` for binary, `0o` for octal, `0x` for hexadecimal, and no prefix for decimal.)

```
print(int("0b101", 0))  # 5
print(int("0×ff", 0))  # 255
```

Further reading:

- Base conversion in Python.

5 – First element that satisfies a condition

When you have an iterable and need to find the first element that satisfies a condition, you can use a generator expression and the built-in `next` to fetch that first element.

The generic recipe looks like this:

```
first = next(elem for elem in iterable if condition(elem))
```

This is a good idea because the generator expression/`next` combo ensures you only search until you find the element you care about. This means that you don't have to compute the condition on the values that come after the value that you wanted.

```
important_numbers = [42, 73, 10, 16, 0]
print(
    next(n for n in important_numbers if n % 2)
)  # 73
```

If there's no such element, you'll either

1. get a `StopIteration` you need to handle:

```
important_numbers = [42, 10, 16, 0]
try:
    print(
        next(n for n in important_numbers if n % 2)
    )
except StopIteration:
    print("No odd numbers found!")
# No odd numbers found!
```

2. pass a default/sentinel value to `next` as its second argument:

```
important_numbers = [42, 10, 16, 0]
print(
    next(
        n for n in important_numbers if n % 2,
        None,
    )
)  # None
```

6 – Last element that satisfies a condition

If you have a condition, you can get the last element of an iterable that satisfies that condition with `collections.deque` and a generator expression:

```
from collections import deque

last = deque(
    (elem for elem in iterable if condition(elem)),
    maxlen=1,
).pop()
```

If there are no such elements, the `deque` will be empty and popping gives an `IndexError`, so you may need to account for that. You can either

1. check if the `deque` has any elements before popping:

```
from collections import deque

important_numbers = [42, 10, 16, 0]
dq = deque((num for num in important_numbers if num % 2), maxlen=1)
if not dq:
    print("No odd numbers found!")
# No odd numbers found!
```

2. or you can handle the `IndexError` that you might get when popping:

```
from collections import deque

important_numbers = [42, 10, 16, 0]
try:
    last_odd = deque(
        (num for num in important_numbers if num % 2),
        maxlen=1,
    ).pop()
except IndexError:
    print("No odd numbers found!")
# No odd numbers found!
```

Further reading:

- Overview of the module `collections`
- `deque` tutorial

7 – Unique elements from a list

The built-in type `set` can be used if you need to compute the unique values from an iterable, like a list.

Because sets are unordered, the result will contain the unique values in an arbitrary order.

```
nums = [42, 73, 42, 42, 0, 73, 10, 10, 16]
for unique_num in set(nums):
    print(unique_num, end=" ")
# 0 73 42 10 16
```

If the order is important and you're running Python 3.8 or later, you can use `dict.fromkeys` instead:

```
nums = [42, 73, 42, 42, 0, 73, 10, 10, 16]
for unique_num in dict.fromkeys(nums):
    print(unique_num, end=" ")
# 0 73 42 10 16
```

These two options are very efficient and only work with hashable values.

Further reading:

- Itertools recipes for `unique_justseen`, `unique_everseen`, and `unique`

8 – Schedule cleanup actions

If you need to clean up resources when your Python program terminates, (for example, disconnect from a server or database), you can use the function `register` from the module `atexit`.

You pass in a function to `register`, and the function you pass it is scheduled to run when your program terminates (even if it terminates because of an exception).

`register` can also be used as a decorator:

```
import atexit

@atexit.register
def cleanup():
    """Clean up program resources."""
    fake_db.close_connection()
    print("All cleaned up!")
```

9 – `map` with multiple arguments

The Python built-in `map` can be used with 2 or more iterable arguments.

The function beig mapped will take one argument from each iterable:

```
bases = [2, 3, 4, 2, 3, 4]
exps = [2, 2, 2, 3, 3, 3]

for num in map(pow, bases, exps):
    print(num, end=" ")
# 4 9 16 8 27 64
```

This can be more convenient to use than a list comprehension/generator expression in some situations:

```
nums = (b ** exp for b, exp in zip(bases, exps))
# vs
nums = map(pow, bases, exps)
```

For a bonus crazy use, here is how to use this to create an infinite stream of perfect squares:

```
from itertools import count, repeat

squares = map(pow, count(), repeat(2))
```

10 – Remove punctuation from a string

Don't use the method `replace` to remove punctuation from a string. Instead, use the method `translate`.

The method `translate` is an efficient and general-purpose method for replacing (or removing) multiple characters in a string simultaneously.

The method `translate` expects a "translation table" argument in a very specific format, but the string class method `maketrans` can build that for us:

```
import string

# print(string.punctuation)  # !"#$%&'()*+,-./:;⟺?@[\]^_`{|}~
punctuation_removal = str.maketrans("", "", string.punctuation)

s = "Hello, world!"
print(s.translate(punctuation_removal))
# Hello world
```

Further reading:

- String `translate` and `maketrans` methods

11 – Count characters in a file

You can use `chain` from the module `itertools` to iterate over the characters of a file, with `chain.from_iterable(f)`.

Pair it with `Counter`, from the module `collections`, and you have a way to count all characters in a file with `Counter(chain.from_iterable(f))`:

```
from collections import Counter
from itertools import chain

with open("/Users/rodrigogs/.zshrc") as f:
    chars = Counter(chain.from_iterable(f))

print(chars.most_common(5))
# [(' ', 583), ('e', 314), ('t', 273), ('o', 264), ('n', 216)]
```

Now… It's unlikely that you'll have to count the characters in a file very often. But this fun example helps you understand what `chain` can do, and `chain` is quite useful!

(**Note**: by default, line endings of the form `\r\n` will get turned into `\n`, so `\r` won't be counted. This may or may not be desirable.)

Further reading:

- Overview of the module `collections`
- Overview of the module `itertools`

12 – Run-length encoding

The module `itertools` has a very funky iterable called `groupby`. If you're imaginative, you can use it for all sorts of things.

One possible use-case is to compute the run-length encoding of an iterable. All it takes is to go through the grouped iterable and then compute the length of each group:

```
from itertools import groupby

def run_length_encoding(iterable):
    for val, group in groupby(iterable):
        yield val, len(list(group))
```

Each group is a lazy iterable iself, so you can't use `len` directly on it. That's why you see `len(list(group))` in the code above.

Here's an example usage:

```
print(list(
    run_length_encoding("AAAB0AA")
))  # [('A', 3), ('B', 1), ('0', 1), ('A', 2)]
```

13 – String prefixes and suffixes

Strings have four convenience methods to replace some slicing: `startswith`, `endswith`, `removeprefix`, and `removesuffix`.

These methods are preferred over the slicing alternatives because they are more convenient and more readable. (The methods `removeX` require Python 3.9+.)

Here are two examples operating on the start of a string:

```
string = "Hello, world!"
print(string.startswith("Hello"))
# True
print(string.removeprefix("Hello"))
# , world!
```

The methods `startswith` and `endswith` also accept a tuple of strings to check:

```
string = "abracadabra"
possible_prefixes = ("aa", "ab", "ac")
print(string.startswith(possible_prefixes))
# True
```

14 – Multiple options in a single `case` statement

Structural pattern matching lets you specify multiple options in the same `case` statement.

You separate the multiple options with a vertical bar `|`. Here is an example:

```
def walk(direction):
    match direction:
        case NORTH | UP:
            return (0, -1)
        ...
```

Further reading:

- Structural pattern matching tutorial

15 – Round to pretty whole numbers

The built-in `round` can be used to round numbers to nice, pretty integers. More concretely, you can use `round` to round numbers to powers of 10.

For example, to round 1374*to*1400 you would do `round(1374, -2)`.

The second argument of `round`, which must be an integer, will give you the power of `0.1` that the number will be rounded to:

| n | 0.1 ** n | | | | | 2 | 0.01 | | 1 | 0.1 | | 0 | 1 | | -1 | 10 | | -2 | 100 |

16 – Type statements

Since Python 3.12 that you can use type statements to create type aliases, which can also be generic.

For example, the statement below creates a type alias called `Pair` that holds pairs of values of the same type:

```
type Pair[T] = tuple[T, T]
```

This is much shorter than the equivalent pre-Python 3.12 code using `typing.TypeAlias` and `typing.TypeVar`:

```
from typing import TypeAlias, TypeVar

T = TypeVar("T")

Pair: TypeAlias = tuple[T, T]
```

For both versions of the code, the following assignment type-checks:

```
p: Pair[int] = (3, 4)
```

Further reading:

- `type` statement and type aliases

17 – Create context managers with `contextlib.contextmanager`

The module `contextlib` provides a decorator `contextmanager` that you can use to implement your own context managers.

For that, you just create a generator that yields once. The code before the `yield` is the setup and the code after the `yield` is the cleanup.

Whatever you yield (if anything) can be captured by the `as ...` part of the `with` statement.

Here is an example that reimplements the built-in `open`:

```
from contextlib import contextmanager

@contextmanager
def my_open(path, mode):
    try:
        file = open(path, mode)
        yield file
    finally:
        file.close()
```

The trick is that the `finally` will ensure we close the file, regardless of whether there is an error while working with the open file.

18 – Immutable dictionary

The module `types` exposes `MappingProxyType`, a type that's essentially an immutable dictionary.

So, if you want an immutable dictionary, create a regular one and wrap it in `MappingProxyType`:

```
from types import MappingProxyType

my_dict = MappingProxyType(
    {
        "url": "mathspp.com",
        "email": "rodrigo@mathspp.com",
    }
)

print(my_dict["url"])  # mathspp.com

# TypeErrors:
my_dict["name"] = "Rodrigo"
my_dict["url"] = ""
```

Be careful not to keep references to the underlying dictionary, though... If you do, and if you modify the underlying dictionary, the changes are reflected in the immutable dictionary:

```
from types import MappingProxyType

base_dict = {
    "url": "mathspp.com",
    "email": "rodrigo@mathspp.com",
}
immutable = MappingProxyType(base_dict)

print(immutable["url"])  # mathspp.com

base_dict["url"] = "example.com"
print(immutable["url"])  # example.com
```

Further reading:

- How to make an Immutable Dict in Python

19 – Self-debugging f-strings

f-strings have an awesome feature: if you include an equals sign `=` at the end of the formatted value, the f-string will show you the code and the value that you're formatting.

Here is an example:

```
name = "RoDrIgO"
print(f"Method title: {name.title() = }")
# Method title: name.title() = 'Rodrigo'
```

Note that the spaces around the equals sign `=` are not necessary but the result usually looks better if you include them.

20 – Dunder attribute `__file__`

The dunder attribute `__file__` can be used to get the full path of your Python script or module.

This can be useful, for example, to locate a resources folder that is "next" to your code in your project directory:

```
from pathlib import Path

print(__file__)
# /Users/rodrigogs/Documents/my_project/example.py

RESOURCES = (Path(__file__).parent / "res").resolve()
print(RESOURCES)
# /Users/rodrigogs/Documents/my_project/res
```

21 – Current date and time

The modue `datetime` has data types that you can use to represent pure dates or dates with times:

1. `datetime.date`
2. `datetime.datetime`

Each class has a class method that gives you an instance of that class with the current date (and time), respectively:

1. `datetime.date.today`
2. `datetime.datetime.now`

```
import datetime as dt

today = dt.date.today()
print(today)  # 2025-04-05

now = dt.datetime.now()
print(now)  # 2025-04-05 19:29:13.437736
```

22 – Set operations with `dict.keys()`

Dictionaries have a method `keys` that returns a view over the keys of the dictionary. These objects support set operations, which means you can manipulate dictionary keys very efficiently and conveniently.

For example, for two dictionaries `dict1` and `dict2`, you can easily compute:

1. the keys available simultaneously in both dictionaries with `dict1.keys() & dict2.keys()`;
2. the keys available in `dict1` but not in `dict2` with `dict1.keys() - dict2.keys()`; and
3. the keys available in either dictionary with `dict1.keys() | dict2.keys()`.

Here are the corresponding examples:

```
en_pt = {  # dict1
    "yellow": "amarelo",
    "red": "vermelho",
}

en_fr = {  # dict2
    "red": "rouge",
    "blue": "bleu",
}
```

```
# Keys in both:
print(en_pt.keys() & en_fr.keys())
# {'red'}

# Keys in en_pt but not in en_fr:
print(en_pt.keys() - en_fr.keys())
# {'yellow'}

# Keys in either:
print(en_pt.keys() | en_fr.keys())
# {'red', 'yellow', 'blue'}
```

23 – Chain multiple dictionaries

You can use the object `ChainMap` from the module `collections` to create a unified view over a hierarchy of dictionaries. The `ChainMap` object accesses the underlying dictionaries in order, stopping once it finds the key you are looking for:

```
from collections import ChainMap

default = {
    "user": "user",
    "theme": "light",
    "lan": "en",
}

local = {
    "theme": "dark",
}

user = {
    "user": "rodrigo",
}

settings = ChainMap(user, local, default)

print(settings["user"])  # rodrigo
print(settings["theme"])  # dark
print(settings["lan"])  # en
```

The underlying dictionaries can still be modified and the changes are reflected in the chained view:

```
user["lan"] = "pt"
print(settings["lan"])  # pt
```

Further reading:

- Module `collections` overview

24 – Longest word in a string

The built-in `max` has a keyword parameter `key` that determines how objects are compared, allowing flexible comparisons.

For example, the idiom `max( ... , key=len)` lets you find the longest item in a collection, namely, the longest word in a string:

```
s = "These are just some sensational words"
print(
    max(s.split(), key=len)
)  # sensational
```

The built-ins `min` and `sorted` also have this keyword parameter.

25 – Dynamic attribute manipulation

The built-ins `getattr`, `setattr`, and `delattr`, can be used to manipulate attributes dynamically.

Whenever possible, you will want to use the dot syntax to access attributes, set attributes, and delete attributes, but these dynamic functions can be used when you have the name of the attribute you want to work with as a string that you computed programmatically.

The built-in `setattr` accepts the object you want to set an attribute on, the attribute name as a string, and the value the attribute will be set to:

```
class Colour:
    pass

c = Colour()
setattr(c, "r", 255)  # c.r = 255
setattr(c, "g", 125)  # c.g = 125
setattr(c, "b", 0)    # c.b = 0
```

The built-in `getattr` accepts the object you want to get an attribute from and the attribute you want to fetch. Typically, you will have the attribute name in a variable:

```
attr = "g"
print(getattr(c, attr))  # 125
print(c.g)               # 125
```

If you use the built-in `getattr` to access an attribute that isn't there, you get an exception `AttributeError`. Alternatively, you can pass in a third argument to `getattr`:

```
print(getattr(c, "x", "heh"))  # heh
```

Finally, the built-in `delattr` will take the given object and delete the attribute specified from it:

```
delattr(c, "g")
print(c.g)  # AttributeError
```

26 – Notify parent class when subclassing

The dunder class method `__init_subclass__` can be used to notify a class when it's subclassed. This is effective for some metaprogramming without having to resort to metaclasses.

In this example, the class `ParentCls` will print a message whenever it is subclassed:

```
class ParentCls:
    def __init_subclass__(cls, **kwargs):
        print(f"{cls} created with {kwargs = }")
```

The argument `cls` will be the subclass, and the keyword arguments `kwargs` come from the subclass definition:

```
class ChildCls(ParentCls, example=True):
    pass
```

When the class `ChildCls` is created, the parent class automatically prints the following:

```
<class '__main__.ChildCls'> created with kwargs = {'example': True}
```

27 – Enforce keyword arguments for options

You can use a single asterisk `*` in a function definition to force all following arguments to be keyword-only.

This is particularly helpful for arguments that act as options or as configuration values. Here is an example with a function that can return the temperature in a room in two units, Celsius and Fahrenheit:

```
def get_temperature(room, *, unit)
```

By using `*`, the second argument must be passed as a keyword argument:

```
get_temperature("bedroom", unit="celsius")  # This works.
```

If you don't, you get an exception `TypeError`:

```
get_temperature("bedroom", "celsius")  # TypeError
```

28 – Flag enumerations

The module `enum` contains a type `Flag` that you can use for enumerations that should support the Boolean operations `&` (AND), `|` (OR), `^` (XOR), and `~` (INVERT):

```
from enum import Flag, auto

class Color(Flag):
    RED = auto()
    GREEN = auto()
    BLUE = auto()

# Purple is red with blue:
purple = Color.RED | Color.BLUE
```

Flag enumerations also support other useful operations, like containment check:

```
# Is the flag GREEN set?
print(Color.GREEN in purple)  # False
```

In Python 3.11+, you can also get a list of the individual flags that are set:

```
# What flags is `purple` composed of?
print(list(purple))  # [<Color.RED: 1>, <Color.BLUE: 4>]
```

29 – Use `Literal` for options

Use the type `Literal` from the module `typing` when a function accepts a small number of specific values that represent configurations or options.

For example, instead of

```
def get_temperature(city: str, unit: str) -> float: ...
```

you can do

```
from typing import Literal

def get_temperature(
    city: str,
    unit: Literal["celsius", "fahrenheit"],
) -> float:
    ...
```

You would still use the function with the plain strings:

```
print(get_temperature("Lisbon", "celsiu"))  # 18.0
print(get_temperature("Lisbon", "fahrenheit"))  # 64.4
```

One of the side-benefits of using the type `Literal` is that you're documenting the valid values.

30 – Return value of a generator

Generators can return a final value once they're finished:

```
def my_generator_function():
    yield 1
    yield 2
    return 73
```

This final value is then attached to the exception `StopIteration` that is raised when the generator is exhausted:

```
gen = my_generator_function()
print(next(gen), next(gen))  # 1 2

next(gen)  # StopIteration: 73
```

You can extract this final value from the attribute `value`:

```
gen = my_generator_function()
print(next(gen), next(gen))  # 1 2

try:
    next(gen)
except StopIteration as err:
    print(err.value)  # 73
```

Useful, for example, if you want your generator to produce some final summary statistics.

31 – Enumerations of string values

You shouldn't use random, loose string values in Python:

```
UP = "UP"
DOWN = "DOWN"

def move(direction: str) -> None:
    if direction == UP:
        print("Going up.")
    elif direction == DOWN:
        print("Going down.")
    else:
        raise ValueError()
```

Instead, you should use `StrEnum` from the module `enum`:

```
from enum import StrEnum

class Direction(StrEnum):
    UP = "UP"
    DOWN = "DOWN"

def move(direction: Direction) -> None:
    if direction == Direction.UP:
        print("Going up.")
    elif direction == Direction.DOWN:
        print("Going down.")
    else:
        raise ValueError()
```

String enumerations let you group strings values together, keeping them organised.

It also helps the IDE provide proper autocompletion when using those values. This is ideal for argument options, for example.

Note: `enum.StrEnum` is only available from Python 3.11 onward. In earlier versions, you can define an enumeration that inherits from `enum.Enum` and `str`:

```
from enum import Enum

class Direction(str, Enum):
    ...
```

32 – Most recently-modified file

Due to flexibility of the built-in `max`, it takes one single line of code to find the most recently-modified file in a directory:

```
from pathlib import Path

folder_to_search = Path("/path/to/folder")
most_recent = max(folder.iterdir(), key=lambda p: p.stat().st_mtime)
print(most_recent)  # /path/to/folder/some_file.txt
```

This works by using the method `stat` that provides access to file statistics and then using the attribute `st_mtime` that contains the time of the last file modification.

This line of code is highly flexible!

Do you want to skip directories and only consider files? In that case, filter with a generator expression:

```
most_recent = max(
    (p for p in folder.iterdir() if p.is_file()),
    key=lambda p: p.stat().st_mtime,
)
```

Do you want the search to be recursive? Then, use `folder.rglob(*)` instead of `folder.iterdir()`.

33 – Normalise strings by removing accents

My name is "Rodrigo Girão Serrão" and the "~" on top of the As are standard in Portuguese… And just like the "~", there are hundreds of other accents and weird marks used by hundreds of other languages!

If you don't want any of it, you can write a short Python function that gets rid of those:

```
import unicodedata

def remove_accents(string):
    return  "".join(
        char for char in unicodedata.normalize("NFD", string)
        if unicodedata.category(char) != "Mn"
    )
```

This function can be useful when writing a "slugify" function, for example:

```
def slugify(string):
    return remove_accents(string).lower().replace(" ", "-")

print(slugify("Rodrigo Girão Serrão"))  # rodrigo-girao-serrao
```

The function `remove_accents` leverages the built-in module `unicodedata`, which provides tools to work with the Unicode standard.

(In case you are wondering, the call `unicodedata.normalize("NFD", string)` separates the accents from the letters:)

```
print(list(unicodedata.normalize("NFD", "äáàãñ")))
# ['a', '̈', 'a', '́', 'a', '̀', 'a', '̃', 'n', '̃']
```

34 – Transpose a list of lists

The built-in `zip` can be used with the splat operator `*` to transpose a list of iterables.

For example, you can go from

```
persons = [["Han", "Solo"], ["Obi-Wan", "Kenobi"], ["Darth", "Vader"]]
```

to

```
firsts = ('Han', 'Obi-Wan', 'Darth')
lasts = ('Solo', 'Kenobi', 'Vader')
```

You just need a simple line of code:

```
firsts, lasts = zip(*persons)
```

If you look closely, this is `zip` undoing what `zip` can do, since you can recreate `persons` by doing `zip(firsts, lasts)`:

```
print(list(zip(firsts, lasts)))
# [('Han', 'Solo'), ('Obi-Wan', 'Kenobi'), ('Darth', 'Vader')]
```

The only thing to note is that `zip` produces tuples, so the original variable `persons` contained a list of lists and the output from the snippet above is a list of tuples.

Further reading:

- Article about `zip`

35 – Inline lists and tuples

The splat operator `*` can be used to inline iterables inside other iterables. Just use the asterisk `*` when writingh out a comma-separated list of values and whatever iterable the asterisk is next to will be "flattened" or unpacked in that position.

This means that using `*iterable` in a comma-separated list will be as if the values from `iterable` had been written explicitly in that place.

Even works with generators!

```
firsts = ("Han", "Obi-Wan", "Darth")

def more_firsts():  # Generator
    yield "Frodo"
    yield "Gandalf"

huge_crossover = [
    "Harry", "Hermione", "Ron",
    *firsts,
    *more_firsts(),
    "Guido",
]

print(huge_crossover)
# ['Harry', 'Hermione', 'Ron',
#  'Han', 'Obi-Wan', 'Darth',
#  'Frodo', 'Gandalf', 'Guido']
```

36 – Typing iterables instead of lists

Setting the type of function arguments to `list` when all you need is to be able to iterate over that value is a mistake:

```
# Why must `files` be a list?!
def create_files(files: list[Path]) -> None:
    for file in files:
        ...
```

Thankfully, it's a mistake that is easy to fix: use `Iterable`:

```
from collections.abc import Iterable  # Python 3.9+

def create_files(files: Iterable[Path]) -> None: ...
```

(In Python 3.8, use `typing.Iterable`. In Python 3.9+, use `collections.abc.Iterable`.)

Using `list` is bad because it prevents you from using tuples, generators, iterables from `itertools`, other collections, etc.

Note: keep in mind that if you need to be able to iterate twice or more over the same iterable, you might want to use `Sequence` instead of `Iterable` because of iterators. Iterators are iterables but they can only be iterated over once.

37 – Multi-dictionary

You can create a multi-dictionary in Python with `collections.defaultdict` and the built-in `list` with `multidict = collections.defaultdict(list)`:

```
from collections import defaultdict

multidict = defaultdict(list)
```

This creates a dictionary that maps every single key to an empty list by default, which is why you use `defaultdict` in the first place:

```
print(multidict["SW"])  # []
print(multidict["LotR"])  # []
```

Then, when you want to "add a value to a key", you instead append to the list mapped to by that key:

```
multidict["SW"].append("Han Solo")
multidict["SW"].append("R2D2")
print(multidict["SW"])  # ['Han Solo', 'R2D2']
```

However, it goes without saying that this is "cheating": the dictionary still maps each key to a single list. You're just leveraging the fact that lists can store multiple values in them to.

Further reading:

- Module `collections` overview

38 – Global enumeration members

The module `enum` has a lot of little-known useful tools. For example, you can use the decorator `enum.global_enum` to automatically export your enumeration members to the global namespace of your module.

This means that you can access enumeration members as `re.MULTILINE` instead of `re.RegexFlag.MULTILINE` (yup, the module `re` uses this!).

The decorator `global_enum` can be used on all types of enumerations; the snippet below applies it to a flag enumeration:

```
from enum import Flag, auto, global_enum

@global_enum
class FilePermissions(Flag):
    READ = auto()
    WRITE = auto()
    EXECUTE = auto()
```

After defining the enumeration, enumeration members can be used as if they were globals:

```
BASE_PERMISSIONS = READ | WRITE
```

Accessing members through the enumeration class still works, though.

Further reading:

- Module `enum` overview

39 – Automatic enumeration values

The module `enum` provides a function `auto` that you can use to automatically generate values for your enumeration members.

The default behaviour is to create successive integers starting at 1 for a standard enumeration:

```
from enum import Enum, auto

class Letter(Enum):
    A = auto()
    B = auto()

print(Letter.A.value)  # 1
print(Letter.B.value)  # 2
```

The function `auto` is also smart enough to specialise appropriately, depending on the type of enumeration.

For flag enumerations, it produces powers of 2 for the flags:

```
from enum import Flag, auto

class Permissions(Flag):
    READ = auto()     # 1
    WRITE = auto()    # 2
    EXECUTE = auto()  # 4

print(repr(Permissions.EXECUTE))
# <Permissions.EXECUTE: 4>
```

For string enumerations, it generates lowercase strings that match the member names:

```
from enum import StrEnum, auto

class Direction(StrEnum):
    NORTH = auto()  # north
    SOUTH = auto()  # south
    ...

print(repr(Direction.NORTH))
# <Direction.NORTH: 'north'>
```

Further reading:

- Module `enum` overview

40 – OS-agnostic line splitting

Do you want to split a piece of text into its lines?

You'd think `text.split("\n")` would do the trick, right..?

It kind of does, but not very well. If you're working with the contents of Windows files, that might not work perfectly because of carriage return characters.

The most robust way to split a string into its lines is with the method `str.splitlines`! If you want to preserve the line-ending characters, use `.splitlines(keepends=True)`.

For example, assume the file `some_file.txt` lives on a Windows machine:

```
windows_string = WindowsPath("some_file.txt").read_text()
```

The lines of this file might be terminated with `"\r\n"` instead of just `"\n"`. If that's the case, using `.split("\n")` will leave the invisible carriage returns:

```
print(windows_string.split("\n"))
# ['This is a\r',
#  'multiline string\r',
#  'from a Windows machine.']
```

Using `.splitlines` fixes that:

```
print(windows_string.splitlines())
# ['This is a',
#  'multiline string',
#  'from a Windows machine.']
```

If you set `keepends=True`, the line-ending characters are left on the lines:

```
print(windows_string.splitlines(keepends=True))
# ['This is a\r\n',
#   'multiline string\r\n',
#   'from a Windows machine.']
```

41 – Longest and shortest

The built-ins `max` and `min` have a keyword argument `key` that lets you change the frame of reference for comparisons.

If you then use `functools.partial` to attach another function to `key`, you essentially build new functions just like LEGOs.

My two favourite examples:

1. `max` + `key=len` builds the function `longest`; and
2. `min` + `key=len` builds the function `shortest`.

```
from functools import partial

longest = partial(max, key=len)
shortest = partial(min, key=len)

words = "This is a truly extraordinary sentence".split()
print(longest(words))  # extraordinary
print(shortest(words))  # a
```

42 – Bounded cache

If you have a deterministic function with no side-effects that gets called very often, consider caching its results. If said function lives in a long-running application (e.g., a web server), make sure you don't run out of memory by ensuring the cache has a maximum sizes.

You can do both of these things with the decorator `functools.lru_cache`, which accepts the cache size as an argument.

For example, `@lru_cache(1024)` in the snippet below creates a cache that saves up to 1024 different call results.

```
from functools import lru_cache

@lru_cache(1024)
def function_to_cache(*args): ...

# Some function calls...

print(function_to_cache.cache_info().currsize)  # 35
print(function_to_cache.cache_info())
# CacheInfo(hits=12, misses=35, maxsize=1024, currsize=35)
```

You can access cache information by using the method `.cache_info` that is added to the function that gets a cache.

43 – Read files in chunks

The built-in `iter` can be used to turn functions into iterables. In its not-so-well-known form, `iter(f, sentinel)` creates an iterable that calls the function `f` until the function returns the value `sentinel`.

For example, paired with `functools.partial`, you can use it to create a "chunk" reader that reads files in chunks:

```
from functools import partial

with open("bee-movie-script.txt", "r") as f:
    chunk_reader = iter(partial(f.read, 16), "")
    for chunk in chunk_reader:
        print(chunk)

"""
According to all
 known laws of a
viation, there i
s no way a bee s
hould be able to
...
"""
```

Further reading:

- Making an iterator out of a function

44 – Format specifier `!r`

When you're using f-strings, you can use the format specifier `!r` to use a value's debugging representation instead of pretty-printing it.

Some values and types cannot be distinguished from one another if you pretty-print them, but can if you use their debugging representation. For example, if a string represents an integer, you can't distinguish it from the same integer when printing:

```
s = "3"
print(f"{s}")  # 3
# !? Was `s` the string "3" or the integer 3?
```

Using `!r` makes it clearer what's being printed:

```
print(f"{s!r}")  # '3'
```

Here's another example:

```
from fractions import Fraction

one_third = Fraction(1, 3)
print(f"{one_third}, {one_third!r}")
# 1/3, Fraction(1, 3)
```

Further reading:

- `str` and `repr`

45 – Counting values that satisfy a predicate

(This is my favourite line of Python code. Really.)

To count the values of an iterable that satisfy a given predicate (a function that returns `True`/`False`) or a given condition, use the built-in `sum` and a generator expression:

```
sum(predicate(value) for value in iterable)
```

This idiom works in 3 parts:

1. the generator expression goes over all values you want to consider;
2. `predicate(value)` evaluates the condition, producing `True` or `False`; and
3. the built-in `sum` accumulates all the Boolean values, effectively counting the number of `True`s.

If you actually have a predicate function, you might prefer the version `sum(map(predicate, iterable))`. If you want to use an ad-hoc expression as the condition, then use the generator expression:

```
ages = [42, 73, 16, 10, 4, 6]

can_vote = sum(age > 18 for age in ages)
print(can_vote)  # 2
```

46 – Dot product idiom

The dot product is a mathematical operation that can be computed with a simple Python idiom using the module `operator`:

```
import operator

sum(map(operator.mul, vec1, vec2))
```

The snippet above assumes `vec1` and `vec2` are iterables that represent vectors. This idiom was present in the documentation of the module `itertools` up to Python 3.11.

Then, in Python 3.12, the built-in `zip` got the keyword argument `strict`, which made the idiom evolve into something that looks a bit more complicated:

```
from itertools import starmap
import operator

sum(starmap(operator.mul, zip(vec1, vec2, strict=True)))
```

We're using `zip( ... , strict=True)` to ensure the vectors have the same size and `zip` produces tuples, so `starmap` is being used to "unpack" that tuple into the two arguments to `operator.mul`.

Then, in Python 3.14, the built-in `map` got a similar keyword argument `strict`, which means the idiom can go back to its simpler form with the extra safety check:

```
import operator

sum(map(operator.mul, vec1, vec2, strict=True))
```

47 – Batching API calls

Since Python 3.12 that the module `itertools` has `batched`: it accepts an iterable and it produces batches of values from that iterable.

You can use this for all sorts of batch processing. For example, you can use `batched` to batch API calls.

Many APIs have ways to handle one request or multiple similar requests at the same time. For example, my newsletter subscriber has an API endpoint that allows me to add tags to a subscriber. However, there is a similar endpoint that allows me to do the same thing to multiple subscribers at the same time.

The second option means I hit the API fewer times, which makes my code run faster. There's less back-and-forth over the network.

Here's the pseudo-code for comparison. First, handling a single user per request:

```
users_to_update = [...]
for user in users_to_update:
    api.add_tag(user, "some tag")
```

Now, the pseudo-code for the batch updates:

```
from itertools import batched

users_to_update = [...]
# The API can't handle more than 50 users at a time.
for user_batch in batched(users_to_update, 50):  # <--
    api.add_tag_to_users(user_batch, "some tag")
```

48 – Redacting email addresses

You can use f-strings and the string formatting specification language to create an effect of redacted or private data. For example, the function below redacts part of your email address:

```
def redact_email(email):
    user, _, domain = email.partition("@")
    return f"{user[:2]:*<{len(user)}}@{domain}"

print(redact_email("rodrigo@mathspp.com"))
# ro*****@mathspp.com
```

The part that is doing the heavy lifting is the section `*<{len(user)}` inside the f-string formatting:

1. `{len(user)}` uses the length of the variable `user` to determine the width of the field where `user[:2]` (the first two characters of the user) will be inserted;
2. `<` tells Python to align `user[:2]` on the left of that field; and
3. `*` tells Python to fill empty space with the character asterisk.

You could modify the function to also mask the domain, for example.

Note: for very security-sensitive use-cases, you might want to randomise the number of asterisks shown, instead of making the string match the correct length.

49 – Random choices

Predictability is usually helpful, but it can also be quite boring. On the other hand, randomness isn't always helpful, but sometimes it's the only way to get something done.

(As a real-world example, the generic profile pictures in the testimonials page on my website have random patterns and random colours.)

To pick a random value from a list, you have two alternatives:

- Use `random.choices` if you want to pick values with replacement (values can be repeated):

```
import random

coin_sides = ["heads", "tails"]

print(random.choices(coin_sides, k=4))
# ['heads', 'tails', 'tails', 'tails']
```

- Use `random.sample` if you want to pick values without replacement (values cannot be repeated):

```
import random

colours = ["red", "green", "blue",
           "black", "white"]
print(random.sample(colours, k=3))
# ['black', 'green', 'blue']
```

For either, set `k` to specify how many values you want to draw from the given list.

Note: for security-sensitive randomness, use the module `secrets`; not the module `random`.

50 – Dynamic regex replacements

The module `re` allows you to do dynamic string replacements. That is, search and replace for certain patterns and then replace them with *other* things that are not constant.

For this, you need a function that accepts a regex match and returns a replacement.

For example, the function `replace` below (admittedly, a poorly-named function), accepts a match and returns a string of asterisks that is as long as the full match:

```
def replace(match):
    return "*" * len(match.group(0))
```

Then, when using `re.sub`, pass it the function that does the replacements instead of specifying a fixed string:

```
import re

text = "I know Python, C, C++, JavaScript, and Haskell."

bad_words = r"C(\+\+)?|JavaScript"

print(re.sub(bad_words, replace, text))
# I know Python, *, **, **********, and Haskell.
```

Further reading:

- Dynamic string replacements with regex blog article

51 – String constants

The module `string` defines many useful constants that you can, and should, use! These will save you the time of defining the constants yourself and prevent plenty of easily-avoidable bugs.

Here are three examples:

```
import string

print(string.ascii_lowercase)
# abcdefghijklmnopqrstuvwxyz

print(string.digits)
# 0123456789

print(string.punctuation)
# !"#$%&'()*+,-./:;⟺?@[\]^_`{|}~
```

If you find that this tip is silly, consider the fact that I found over 10,000 repositories on GitHub with bugs because they had typos when defining constants with the full latin alphabet. These bugs could have been avoided by using `string.ascii_lowercase`…

Further reading:

- Finding and fixing over 10,000 bugs on GitHub blog article.

52 – Case-insensitive regular expressions

Regular expressions can start with the flag `(?i)`, marking them as case-insensitive:

```
import re

print(re.match(r"hey", "HeY"))
# None

print(re.match(r"(?i)hey", "HeY"))
# <re.Match object; span=(0, 3), match='HeY'>
```

53 – Module `itertools` categorisation

A good way to think about the module `itertools` is to remember it has five categories of iterables:

1. reshaping iterables: `batched`, `chain`, `groupby`, `islice`, and `pairwise`;
2. filtering iterables: `compress`, `dropwhile`, `filterfalse`, and `takewhile`;
3. combinatorial iterables: `combinations`, `combinations_with_replacement`, `permutations`, and `product`;
4. infinite iterables: `count`, `cycle`, and `repeat`; and
5. tool-complementing iterables: `accumulate`, `starmap`, and `zip_longest`.

Knowing these five categories should help you remember what tools you have available.

(There's also `tee` in `itertools`, which manipulates iterables but isn't an iterable itself!)

Further reading:

- Module `itertools` overview blog article.

54 – t-strings need processing

t-strings were introduced in Python 3.14.

t-strings are a generalisation of f-strings that let you control the formatting process a bit more. Their main use case is to allow for safer formatting when you're formatting user input that needs to be sanitised. (C.f. the ever-relevant xkcd comic 327.)

It is up to the programmer to call a function that takes a t-string and processes its interpolated values. In the example below, the programmer needs to use a function `interpolate_html_safe` to interpolate HTML safely, so as to avoid potential security issues arising from including arbitrary JavaScript in the final HTML:

```
from string.templatelib import Template

def interpolate_html_safe(template: Template) -> str:
    ...  # Processes interpolated values ...

to_format = "<script>alert('Malicious JS');</script>"
html_page = t"<html>{to_format}</html>"
print(interpolate_html_safe(html_page))
# <html>&lt;script&gt;alert('Malicious JS');&lt;/script&gt;</html>
#       ^^^^      ^^^^                       ^^^^       ^^^^
# Script tags were escaped when interpolating the string.
```

55 – Structural unpacking

When doing an assignment, if the value on the right is an iterable (list, tuple, …), you can unpack it. On the left, you can write as many variables as elements you expect to have on the right. You can also use the splat operator to capture lists of zero or more elements.

And you can nest these structural matches!

```
colour = ("AliceBlue", (240, 248, 255, 255))
name, (*rgb, alpha) = colour

print(rgb)  # [240, 248, 255]
```

This also works with the assignment target in a `for` loop:

```
colours = [colour, ...]

for name, (*rgb, alpha) in colours:
    print(name, rgb, alpha)

# AliceBlue [240, 248, 255] 255
# ...
```

Further reading:

- Structural unpacking blog article.

56 – Ergonomic multiline strings

When I'm writing multiline strings I like to have the `"""` by themselves, for readability. However, this creates an extra empty line at the beginning and end of the string:

```
string = """
Multiline string.
No escaped newlines
"""

print("> " + string + "!")
```

```
>
Multiline string.
No escaped newlines
!
```

This is not what I want… To fix this, I can use the backslash character `\` to escape those extra newlines:

```
string = """\
Multiline string.
First & last newlines escaped\
"""

print("> " + string + "!")
```

```
> Multiline string.
First & last newlines escaped!
```

57 – Underscore in the REPL

When working in the REPL, the result of the last non-`None` expression is saved in the special variable `_` (underscore).

This is especially useful if you run a slow piece of code and forget to assign the result. Just do `result = _`.

As someone who uses the REPL a lot, I find this to be very helpful!

Here is an example REPL session showcasing this feature:

```
>>> 3 ** 3 ** 3
7625597484987
>>> print(_)
7625597484987

>>> _
7625597484987

>>> sum([_, _, -_, -_])
0
>>> _
0
```

Note that functions that return `None` do not update the value stored in `_`.

Further reading:

- The appearing built-in.
- Usages of the underscore.

58 – Subclassing immutable types

How do you subclass immutable types? (For example, how would you subclass floats?)

The dunder method `__init__` alone isn't enough; you need something else…

You need the dunder method `__new__`, `__init__`'s big brother. The dunder method `__new__` is a class method that is responsible for creating the object, whereas `__init__` simply initialises/customises it.

Here is the skeleton for a float subclass:

```
class FloatSubclass(float):
    def __new__(cls, value, *args, **kwargs):
        print("__new__", value, args, kwargs)
        return super().__new__(cls, value)

    def __init__(self, value, *args, **kwargs):
        print("__init__", value, args, kwargs)
        # Do whatever with the args and kwargs

x = FloatSubclass(4.5)
# __new__ 4.5 ('hello',) {'foo': True}
# __init__ 4.5 ('hello',) {'foo': True}
print(x)  # 4.5 <- looks like a float.
```

Further reading:

- Customising object creation with `__new__`.

59 – Idiomatic sequence slicing

There are five slicing patterns that are fairly common and that have simple interpretations, and that's what an idiom is: a piece of code that you recognise for its meaning as a whole.

You should be able to interpret these five slicing idioms automatically without having to think:

1. `[:n]` – first `n` chars
2. `[n:]` – after first `n` chars
3. `[:-n]` – without last `n` chars
4. `[-n:]` – last `n` chars
5. `[::-1]` – reversed (but the built-in `reversed` is usually preferred)

```
string = "Slicing is easy!"

print(string[:4])  # Slic
print(string[4:])  # ing is easy!

print(string[:-4])  # Slicing is e
print(string[-4:])  # asy!

print(string[::-1])  # !ysae si gnicilS
```

Further reading:

- Idiomatic sequence slicing blog article.

60 – File tail

You can get the last few lines of a file with `collections.deque`. You just have to set `maxlen` to the number of lines you want from the end of the file:

```
from collections import deque

with open("/path/to/python/lib/this.py", "r") as f:
    tail = deque(f, maxlen=4)

for line in tail:
    print(line)

#     for i in range(26):
#         d[chr(i+c)] = chr((i+13) % 26 + c)
#
# print("".join([d.get(c, c) for c in s]))
```

If the file has fewer lines than the ones you asked for, you get the full file.

Further reading:

- `deque` tutorial.

61 – One-shot file I/O

For one-shot file I/O, like simply reading the full contents of a file or writing some text to a file, you can use the methods `read_text` and `write_text` from `pathlib.Path`.

Here is a short example:

```
from pathlib import Path

filepath = Path("hello_world.txt")

filepath.write_text("Python is cool!")

print(filepath.read_text())
# Python is cool!
```

Note that if the file exists, `write_text` will overwrite the contents of the file.

If you use a context manager and the built-in `open` you get more control, but these two methods get the job done most of the time.

Further reading:

- Module `pathlib` overview.

62 – Formatting big numbers

When doing string formatting with big integers, you may want to include thousands separators to make numbers easier to read. You can add

- commas;
- underscores; or
- a locale-appropriate separator.

To do so, use the specifiers `,`, `_`, or `n`, respectively:

```
bignum = 123541241234

print(f"Big money ${bignum:,}")
# Big money $123,541,241,234

print(f"Big money ${bignum:_}")
# Big money $123_541_241_234

print(f"Big money ${bignum:n}")
# Big money $123541241234  # Might be different for you.
```

Note that the locale-appropriate separator might be "nothing" if the locale isn't set appropriately beforehand, as seen above.

Since Python 3.14, you can also *group digits in the fractional part* of a number.

Further reading:

- Thousands separators.

63 – Named groups in regex

If you're using regex (I'm sorry!) and you want to extract portions of patterns with groups, you can use named groups to make extraction easier.

To create a named group, start the group with `?P< ... >` and put the group name inside the angled brackets `<>`, as the example below shows:

```
import re

pattern = r"(?P<user>\S+)@(?P<domain>\S+)"
```

Then, when you get a match, you can use the method `group` to extract groups by name:

```
match = re.match(pattern, "rodrigo@mathspp.com")

print(match.group("user"))  # rodrigo
print(match.group("domain"))  # mathspp.com
```

Note that this is a Python-specific feature and that are unlikely to have this feature when working with regex in other contexts/programming languages.

64 – Resolving paths

When working with paths, and especially if user input is involved, you often want to normalise and resolve your paths so you have absolute paths that are expanded.

For example, you might need to turn

- something `"~/Documents"` into `"/Users/rodrigogs/Documents"` (or whatever the equivalent is in your OS); or
- something like `"~/Documents/../Downloads"` into `"/Users/rodrigogs/Downloads"`, getting rid of the `~` and the `...`.

To expand the `~` into your home folder, use the method `expanduser`:

```
from pathlib import Path

docs = Path("~/Documents")
abs_docs = docs.expanduser()
print(abs_docs)  # /Users/rodrigogs/Documents
```

To resolve references with `..` or `.`, use the method `resolve`:

```
downloads = abs_docs / ".." / "Downloads"
print(downloads)  # /Users/rodrigogs/Documents/../Downloads
print(downloads.resolve())  # /Users/rodrigogs/Downloads
```

To do everything in one go, use `Path(filepath).expanduser().resolve()`. And make sure you call `expanduser` first, since calling `resolve` first might lead to the wrong result:

```
downloads = docs / ".." / "Downloads"
print(downloads)  # ~/Documents/../Downloads

# Correct order:
print(downloads.expanduser().resolve())  # /Users/rodrigogs/Downloads

# Wrong order:
print(downloads.resolve().expanduser())  # /Users/rodrigogs/Documents/~/Downloads
```

65 – Formatting dates with f-strings

Dates, times, and datetime objects, from the module `datetime`, can be formatted directly in f-strings.

You don't need to use `strptime`. Or `strftime`, whichever one of these two does the formatting.

The format specifier can contain the special codes with `%` to refer to parts of the date but anything else is left untouched.

```
from datetime import date

print(f"{date.today():%Y-%m-%d}")
# 2025-06-03

print(f"{date.today():%Y——%m :: %d}")
# 2025——06 :: 03
```

Further reading:

- Datetime objects and f-strings.

66 – Concatenate files from handlers

Suppose you need to concatenate multiple files together, for example to write a long file composed of multiple other files or to search for a pattern across all files.

If you *already* have the file handlers in a list, use `chain.from_iterable` to go through the whole thing.

This is as memory efficient as possible since it only reads the lines from each file on demand and it doesn't hold the full contents of each file in memory.

For example, suppose you have three files:

1. `log1.log`:

```
08:12:03 [INFO] Starting the data sync process
08:12:04 [DEBUG] Loaded 142 user records from cache
```

2. `log2.log`:

```
08:12:06 [INFO] Connection to remote server established
08:12:08 [WARN] Response time exceeded threshold: 534ms
```

3. `log3.log`:

```
08:12:09 [DEBUG] Retry attempt #1 initiated
08:12:11 [INFO] Data sync completed successfully
```

If the files are already open in the variables `log1`, `log2`, and `log3`, respectively, then you can run

```
open_files = [log1, log2, log3]

from itertools import chain

with open("full_log.log", "w") as f:
    f.writelines(chain.from_iterable(open_files))
```

This produces the file `full_log.log` with the contents of all three files:

```
08:12:03 [INFO] Starting the data sync process
08:12:04 [DEBUG] Loaded 142 user records from cache
08:12:06 [INFO] Connection to remote server established
08:12:08 [WARN] Response time exceeded threshold: 534ms
08:12:09 [DEBUG] Retry attempt #1 initiated
08:12:11 [INFO] Data sync completed successfully
```

If the files haven't been opened yet, then the best thing to do is to open each one of them at a time, of course.

67 – Generator recipe

Generators are an awesome Python feature because their potential upside in terms of memory and time savings is infinite.

If you don't have experience writing generators, it's actually "easy" with this little trick.

First, you write a function that builds a list with the elements that you want to produce. Then, you follow these three steps:

1. Replace all calls to `.append(value)` with `yield value`.
2. Delete the line where you initialised the list with the results.
3. Delete the line where you return the list with the results.

That's it!

For example, start with this function:

```
def squares(stop):
    result = []
    for n in range(stop):
        result.append(n ** 2)
    return result

print(squares(5))  # [0, 1, 4, 9, 16]
```

Then, you should get this generator:

```
def squares(stop):
    # result = []
    for n in range(stop):
        yield n ** 2  # result.append(n ** 2)
    # return result

print(list(squares(5)))  # [0, 1, 4, 9, 16]
```

This trick also works for infinite generators if you start by writing a loop that builds an infinite list. The function won't work because you can never build an infinite list, but the 3-step process can be applied and the resulting generator works.

68 – Split strings in two halves

Sometimes you'll need to split the string in two halves. When that's the case, you'll want to use the method `partition`:

```
email = "rodrigo@mathspp.com"
user, at, domain = email.partition("@")

print(user)  # rodrigo
print(at)  # @
print(domain)  # mathspp.com
```

If the split fails, the first item of the 3-item tuple contains the full string and the two other items are the empty string:

```
email = "mathspp.com"
user, at, domain = email.partition("@")

print(user)  # mathspp.com
print(at == domain == "")  # True
```

This is better than using `.split( ... , maxsplit=1)` because the return type makes it easier to check if the split was successful. With `partition`, it's enough to check if the middle element is truthy or not:

```
email = input(" >>> ")
user, at, domain = email.partition("@")

if at:
    print("Split successful!")
else:
    print("Split was not successful!")
```

If the separator occurs 2+ times, `partition` splits on the first. Use `rpartition` to split on the last:

```
print("aaa oi bbb oi ccc".partition("oi"))
# ('aaa ', 'oi', ' bbb oi ccc')

print("aaa oi bbb oi ccc".rpartition("oi"))
# ('aaa oi bbb ', 'oi', ' ccc')
```

69 – Regex multiline flag

Regular expressions have a multiline flag that can be used to change the behaviour of the special characters `^` and `$`. By default, `^` and `$` match the beginning and end of the string, respectively; with the multiline flag, they match the beginning and end of each line, respectively.

The flag can be used inline with `(?m)`, or passed in to the relevant functions of the module `re` with `re.MULTILINE`.

As an example, take the following string that spans across two lines:

```
The quick brown fox jumps over
the lazy dog.
```

The pattern `r"^[Tt]he"` will only match the first "The", while the pattern `r"(?m)^[Tt]he"` will match the two occurrences of the word "the".

70 – Extract assignments from conditionals

Instead of using a conditional to make an assignment, use a conditional expression. This extracts the important operation (the assignment) out of the nesting of the `if: ... else: ...` statement, making it easier to spot the relevant part of the code.

If you're not used to conditional expressions, it's a matter of learning how to read them in English (or in your language). The snippet below, using a conditional statement, reads "If the user is logged in, set their permissions to `"full"`, otherwise set them to `"guest"`.":

```
if logged_in:
    permissions = "full"
else:
    permissions = "guest"
```

If you use a conditional expression, it now reads as "Set the user permissions to `"full"` if the user is logged in, otherwise set them to `"guest"`.":

```
permissions = "full" if logged_in else "guest"
#^^^^^^^^^^^^^^^^^^^^                          Set permissions to `"full"`
#                     ^^^^^^^^^^^^             if the user is logged in,
#                                 ^^^^^^^^^^^^ otherwise set them to `"guest"`.
```

Further reading:

- Conditional expressions.

71 – File discovery by name pattern

When you need to find files based on a name pattern, you can use the method `glob` from `pathlib.Path`. Note that `glob` doesn't support complex (regex) patterns, though.

Suppose the filesystem contains the following folder `Downloads` and its given contents:

```
= Downloads/
  - cat_and_dog.png
  - cute_cat.jpg
  - cute_dog.jpg
  - two_cats.png
```

Then, the snippet below finds all files with the string "cat" in the name:

```
from pathlib import Path

cat_files = Path("Downloads").glob("*cat*")
# Finds cat_and_dog.png, cute_cat.jpg, two_cats.png
```

Similarly, the snippet below finds all files that end with ".jpg":

```
from pathlib import Path

jpg_imgs = Path("Downloads").glob("*.jpg")
# Finds cute_cat.jpg, cute_dog.jpg
```

Note that the method `glob` produces the file paths in an arbitrary order.

Further reading:

- Module `pathlib` overview.

72 – Structural pattern matching with dictionaries

The `match` statement can be used to match keys and values in a dictionary, effectively matching the structure of the dictionary.

The example below matches all dictionaries that contain a key `"user"` with a string value and a key `"age"` with an integer value:

```
match my_dict:
    case {"user": str(), "age": int()}:
        print("Match!")
```

Dictionaries that satisfy those restrictions will match, even if they have more key/value pairs, like the dictionary `my_dict` below:

```
my_dict = {"user": "John", "age": 47, "children": 2}
```

The dictionaries below won't match, respectively because they're missing the key/value pair for `"age"` or because the value for `"age"` doesn't have the correct type:

```
my_dict = {"user": "John"}  # Missing `"age"`
my_dict = {"user": "John", "age": "47"}  # `"age"` should be an integer.
```

Further reading:

- Structural pattern matching tutorial.
- Structural pattern matching cheatsheet.

73 – Slicing generators for debugging

When debugging with generators, you may want to use `itertools.islice`. This allows you to easily take a look at the first few elements of the generator.

You can wrap the call to `islice` in `print(list( ... ))` to print the elements, or you can loop over the slice and add a breakpoint immediately before.

Here is an example using `print(list(islice( ... , n)))` to print the first `n` elements immediately:

```
from itertools import islice

gen = mysterious_generator()

print(list(islice(gen, 3)))  # [0, 1, 4]
```

Remember that if you use `islice` again, you'll be slicing from the *remainder* of the generator:

```
print(list(islice(gen, 3)))
# [9, 16, 25]
print(list(islice(gen, 3)))
# [36, 49, 64]
```

74 – Peek at an iterable

To know the first element of a list, you index with `[0]`. However, not all iterables can be indexed.

In general, to get the first element of an iterable (to "peek" at it), you can use `next` to fetch that element and then you can use `chain` from the module `itertools` to "glue it back".

Here is an example of how to do this:

```
from itertools import chain

def peek(iterable):
    iterator = iter(iterable)
    first = next(iterator)
    return first, chain([first], iterator)
```

This specific implementation raises a `StopIteration` error if the iterable is empty. Other behaviours can be easily implemented.

75 – Match the structure of custom objects

Custom objects can have their structure matched by specifying the relevant attribute names in the dunder attribute `__match_args__`.

This dunder attribute must be a tuple of strings that specifies the attribute names used for matching. The object can have more attributes, but the ones that aren't included in `__match_args__` are irrelevant for the purposes of structural pattern matching.

For example, the class `Point` below has three attributes but the dunder attribute `__match_args__` only refers two:

```
class Point:
    __match_args__ = ("x", "y")

    def __init__(self, x, y, z):
        self.x, self.y = x, y
        self.z = z

p = Point(1, 2, 3)
```

When trying to match the structure of an instance of `Point`, you can match directly against the attributes `x` and `y`:

```
p = Point(1, 2, "")
match p:
    case Point(1, int()):
        print("match!")
# match!
```

Note how the case statement with the pattern `Point(1, int())` *looks* like an instantiation of the class `Point`, although that is not valid code for the purposes of class instantiation. It is valid code for the purposes of structural pattern matching, though.

Further reading:

- Structural pattern matching tutorial.
- Structural pattern matching cheatsheet.

76 – Built-in `next` with a default value

The built-in `next` can be given a second argument which is the default return value used when the given iterator is empty. A default value (of `None`, for example) tends to be more useful in idioms that use next. (For example, see *this tip about the first element of an iterable that satisfies a condition*.)

As an example, consider the generator expression assigned to `squares` and that is exhausted by calling the built-in `next` repeatedly:

```
squares = (x ** 2 for x in range(3))

print(next(squares))  # 0
print(next(squares))  # 1
print(next(squares))  # 4
```

The next time that the built-in `next` is called, Python raises a `StopIteration`:

```
print(next(squares))
# StopIteration
```

However, if you pass a second argument to the built-in `next`, that's the result you get:

```
print(next(squares, "hey"))
# hey
```

77 – Match an exact dictionary structure

Structural pattern matching can be used to match dictionaries with given keys and values.

By default, patterns with dictionaries only determine keys and values that a dictionary *must* contain to match. This means dictionaries can have more keys/values than the ones listed, and they still match.

If you wish to match a specific, restricted dictionary structure (that is, if you want to disallow dictionaries from having more key/value pairs), you can use a guard.

For example, consider the following `match` statement:

```
match d:
    case {"name": str()}:
        print("match!")
```

The two dictionaries below would match, even though the second one has a key `"age"` that is not mentioned in the pattern above.

```
{"name": "John"}  # matches
{"name": "John", "age": 42}  # matches
```

To only match dictionaries that have exactly that structure, introduce a guard with `**kwargs` that asserts that there are no extra keys:

```
match d:
    case {"name": str(), **kwargs} if not kwargs:
        print("match!")
```

With this guard, the longer dictionary will not match.

Further reading:

- Structural pattern matching tutorial.
- Structural pattern matching cheatsheet.

78 – Undoable iterator with value history

You can use a `deque` from the module `collections` to create an undoable iterator that keeps track of the history of seen values.

You use a `deque` to hold the history of values that have been seen and also create a `deque` to hold a queue of values that have been seen already but that were "undone".

To fetch the next element from the iterator, ensure there's at least one value in the queue – which you might need to fetch from the original iterable – and then produce that.

To undo a step, you take one element from the history `deque` and plop it into the queue `deque`.

This is a sample implementation of the iterator `undoable` with no validation/error-checking:

```
from collections import deque

class undoable:
    def __init__(self, iterable, hist_size=100):
        self.iterator = iter(iterable)
        self.history = deque(maxlen=hist_size)
        self.queued = deque(maxlen=hist_size)

    def __next__(self):
        if not self.queued:
            self.queued.append(next(self.iterator))
        item = self.queued.popleft()
        self.history.append(item)
        return item

    def undo(self):
        self.queued.appendleft(self.history.pop())

    def __iter__(self): return self  # necessary for the iterator protocol
```

The snippet below shows an usage example with a generator expression:

```
squares = undoable(x ** 2 for x in range(3))

print(next(squares))  # 0
print(next(squares))  # 1

squares.undo()
squares.undo()

print(next(squares))  # 0
print(next(squares))  # 1
print(next(squares))  # 4
```

79 – Typing overloads

The usage of typing overloads allows you to declare relationships between the types of arguments and/or the return type of a function. This allows the type checker to use information that otherwise couldn't be inferred from the function signature alone.

For example, the computation `2 * arg` works for strings and ints, and you know the result has the same type as the value used.

Wrapping that multiplication in a function and adding type hints in the naive way doesn't tell the type checker that there is a relationship between the argument type and the return value:

```
def double(arg: str | int) -> str | int:
    return 2 * arg

reveal_type(double("hey"))  # str | int
```

Note how the type checker doesn't know whether the returned value is a string or an integer, although we know that it will be the string `"heyhey"`.

However, if you use overloads, you can specify that the argument type is the same as the return value. Typing overloads can be as many as needed and they're decorated with `typing.overload`. You only need to specify the function signature with the restricted relationships. Then, you provide the full function signature and its body:

```
from typing import overload

# If you give it strings, you get strings:
@overload
def double(arg: str) -> str: ...

# If you give it integers, you get integers:
@overload
def double(arg: int) -> int: ...

# The general function with the function body:
def double(arg: str | int) -> str | int:
    return 2 * arg

reveal_type(double("hey"))  # str
```

Note how the type checker knows that the return type is `str` because the argument is of the type `str`.

80 – Bulk renaming files

To change the name of a file while preserving its extension you can use the method `pathlib.PurePath.with_stem`. Note that this creates a new path object and doesn't do any file renaming automatically. That is why the method `with_stem` is defined in `PurePath`, although it is also available from `Path`.

If you pair `with_stem` with `rename`, you can easily bulk-rename files in a folder.

Suppose the folder `reports` contains multiple reports in different formats, like PDF files, Excel files, and the occasional screenshot(!). To rename all files, you could write something like

```
from pathlib import Path

report_folder = Path("reports")  # PDFs, Excel files, ...

for idx, report in enumerate(report_folder.iterdir(), start=1):
    report.rename(report_pdf.with_stem(f"report{idx:03}"))

# Produces:
# reports/report001.pdf
# reports/report002.xlsx
# ...
```

Further reading:

- Module `pathlib` overview.

81 – Structural validation and homogenisation

Some functions can accept arguments in many different formats. In that case, structural pattern matching can be used to validate the argument and to homogenise it.

For example, padding in CSS can be specified as a single integer, or as a tuple of 1, 2, or 4 integers. But in CSS the padding is always unwrapped into 4-item tuples.

Some valid examples and the conversions:

- `10` is converted to `(10, 10, 10, 10)`
- `(5, 10)` is converted to `(5, 10, 5, 10)`
- `(1, 2, 3, 4)` is converted to `(1, 2, 3, 4)`

Using the `match` statement you can easily write a function that validates this structure and does the conversion:

```
def unpack_padding(pad) -> tuple[int, int, int, int]:
    match pad:
        case int(p) | (int(p),):
            return (p, p, p, p)
        case (int(vert), int(horz)):
            return (vert, horz, vert, horz)
        case (int(top), int(right), int(bottom), int(left)):
            return (top, right, bottom, left)
        case _:
            raise ValueError(f"1, 2 or 4 integers required for padding; got {pad!r}.
↪")
```

Further reading:

- Structural pattern matching tutorial.
- `match` statement cheatsheet.

82 – Verbose regular expressions

This is the best thing you can do for your regular expressions: use the verbose flag. Verbose regular expressions can be split across multiple lines and you can add comments to it. This makes it much easier to write readable regular expressions.

As an example, here is a not-so-complex regular expression:

```
pattern = r"---\ntitle: (?P<title>.*?)\nauthor: (?P<author>.*?)\n——"
```

Here is an equivalent regular expression after using the verbose flag `(?x)`:

```
pattern = r"""(?x)
    ---\n                          # frontmatter delimiter
    title:\ (?P<title>.*?)\n      # book title
    author:\ (?P<author>.*?)\n    # author name
    ---                            # frontmatter delimiter
"""
```

Note that, since whitespace is ignored, you need the explicit `"\n"` to match newlines and you also need to escape spaces with a backslash `\`.

The beauty of this example is that the verbose regular expression now looks much more like the text that it matches, for example:

```
---
title: Moby Dick
author: Herman Melville
---
```

The verbose flag can also be used as `re.X` or `re.VERBOSE` when passing flags to the functions of the module `re`.

83 – Extracting text data into a dict

Regular expression matches from the module `re` have a method `groupdict` that allow you to create a dictionary with the named groups that your regular expression defines.

This is useful, for example, if you're extracting data from structured text and want to convert it to a more convenient format (a dictionary).

Suppose you have a number of files for copyright-free books with a frontmatter header and the markdown contents:

```
---
title: Moby Dick
author: Herman Melville
---

[...]
```

You can define a regex pattern to extract the title and author information:

```
pattern = r"""(?x)
    ---\n
    title:\ (?P<title>.*?)\n
    author:\ (?P<author>.*?)\n
    ---"""
```

Then, you can use the module `re` and any of its functions to search/find text. If you get a match, you can use the method `groupdict` to create a dictionary with key/value pairs for every *named group* you defined. In the example above, that would be a dictionary with keys `"title"` and `"author"`:

```
import re

print(
    re.match(pattern, text).groupdict()
)
# {'title': 'Moby Dick',
#  'author': 'Herman Melville'}
```

Further reading:

- `re.Match.groupdict`.

84 – Generics syntax

Python 3.12 introduced many typing improvements, namely an improved syntax to define generic functions and generic classes. Instead of having to define type variables beforehand, the type variables needed for the generic function/class can be defined with the object, making it more concise.

For example, instead of

```
from typing import TypeVar

T = TypeVar("T")

def into_list(value: T) -> list[T]:
    return [value]
```

you can use `[ ... ]` after the function name, and before the parameter list, to specify the type variables required:

```
def into_list[T](value: T ) -> list[T]:
    return [value]
```

Similarly, you can define a generic class over a type variable `T` by using `[T]` immediately after the class name:

```
class MyList[T]:
    def __init__(self, values: list[T]):
        self.values = list(values)

    def append(self, value: T ) -> None:
        self.values.append(value)

    def pop(self) -> T :
        return self.values.pop()
```

85 – Add lists together, fast

Adding (concatenating) two lists together is a "slow" operation because it requires creating a new list and "copying" the values from the two source lists to the third resulting list. In performance-sensitive scenarios, it may be better to use `itertools.chain` to chain the two lists together, instead.

For example, suppose there's a scenario where you have a node from a tree-like structure and you need to traverse the tree up, from the current node up until the root. You might write a `for` loop like the one below:

```
for node in [this_node] + list(this_node.parents):
    pass
```

If this were performance-sensitive code, you'd be better off using `chain`:

```
from itertools import chain

for node in chain([this_node], this_node.parents):
    pass
```

The second alternative is faster because:

1. `this_node.parents` doesn't need to be converted to a list; and
2. chaining doesn't copy values to a third container.

This piece of advice is generally applicable but if you're writing truly performance-sensitive code, don't forget to benchmark this change.

Further reading:

- Module `itertools` overview.
- CPython pull request #112406.

86 – File modes

There are 16 different file modes. The four base modes are

1. `r` for reading;
2. `w` for writing;
3. `a` for appending; and
4. `x` for exclusive creation and writing.

The mode `x` is like `w`, but fails if the file already exists. In other words, it ensures you're creating a new file.

The table below encodes the properties of each mode:

Mode	`r`	`w`	`a`	`x`
allows read	x			
allows write		x	x	x
must exist	x			
must not exist				x
positioned at start	x	x		x
positioned at end			x	
always writes to end			x	

These four base modes operate on files in text mode, by default, which is the same as appending a `t` to each mode. For example, the modes `r` and `rt` are the same. Instead, you can append a `b` to open the files in binary mode.

Finally, you can also append a `+` to any mode to enable writing and reading at the same time, although this is uncommon because it can be quite confusing.

Further reading:

- File modes in Python.

87 – Caching sets and frozen sets

The built-in type `set` has a variant `frozenset` that is immutable and that you can use as a dictionary key, for caching purposes, or in other contexts where you require a hashable value.

As an example, suppose you have a function `products_with_tags` that searches for products in a store that have the given tags, and since the database of products is fairly stable, you want to cache these lookups.

You write the function with a cache:

```
from functools import cache

@cache
def products_with_tags(tags):
    ...  # Send an API request,
    ...  # process the results, etc.
```

Since sets aren't hashable, you can't call this function with an argument of the built-in type `set`:

```
tags = {"sale", "local"}
print(products_with_tags(tags))  # Exception
```

Since frozen sets are immutable, they can be made hashable, and thus they can be used here:

```
tags = frozenset({"sale", "local"})
print(products_with_tags(tags))
```

Frozen sets support all operations that sets support and, in particular, they support very fast lookups/containment checks.

The only operations that frozen sets do not support are the set operations that modify the set in-place.

Further reading:

- Sets and frozensets.

88 – Constrained generics

Generics (functions and classes) can be constrained. This means that the variable type must be one of two (or more) specified types.

Consider the function `double` as a motivating example:

```
def double[T](value: T) -> T:
    return 2 * value
```

The fact that the function `double` is generic explicitly states the relationship between the argument type and the return type, which is preserved in calls like `double(2)` or `double("hello")`.

However, with the given type hints, it's not immediately obvious to type hinters what types of values can be passed to the function `double`. To make that explicit, the type variable `T` can be annotated with a tuple of two or more types, therefore constraining `T` to be one of those types:

```
def double[T: (str, int, float)](value: T) -> T:
    return 2 * value
```

In this new definition, type checkers will only accept arguments of type `str`, `int`, or `float`, for the function `double`.

For example, if `None` is used, mypy will complain:

```
Value of type variable "T" of "double" cannot be "None"  [type-var]
```

89 – Natural alphabetical sorting

Strings can be sorted in Python and the default behaviour is to sort lexicographically, character by character. However, characters are compared by their Unicode codepoint (which you can check with the built-in `ord`), which means upper case letters and lower case letters are separated because their codepoints are "far" apart:

```
fruits = ["banana", "APPLE", "Coconut"]
print(sorted(fruits))
# ['APPLE', 'Coconut', 'banana']
```

The lower case letters come after the upper case letters because the lower case letters have higher codepoints:

```
print(ord("C"))  # 67
print(ord("b"))  # 98
```

If you want to sort a series of strings in a case-insensitive way, you can use the built-in `sorted` and set the keyword parameter `key` to the string method `str.casefold`:

```
fruits = ["banana", "APPLE", "Coconut"]
print(sorted(fruits, key=str.casefold))
# ['APPLE', 'banana', 'Coconut']
```

Further reading:

- How to work with case-insensitive strings

90 – Preserving decorated function metadata

The standard way of writing a decorator involves creating a wrapper function for the decorated function, like so:

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        ...  # Decorator code goes here.
    return wrapper
```

Following this strategy has the drawback of "erasing" important function metadata.

Suppose you apply the decorator `my_decorator` to a function:

```
@my_decorator
def add(a, b):
    """Performs addition!"""
    return a + b
```

In doing so, the metadata of the function `add` was messed up. Printing the function now shows a funky result:

```
print(add)  # <function my_decorator.<locals>.wrapper at 0×102cb4510>
```

Similarly, using the built-in `help(add)` will reveal a cryptic help message:

```
Help on function wrapper in module __main__:

wrapper(*args, **kwargs)
 Help on my_decorator.<locals>.wrapper line 1/3 (END) (press h for help or q to␣
 ↪quit)
```

To fix this behaviour – which is technically correct but definitely unhelpful – you can use the decorator `functools.wraps` inside your decorator:

```
import functools

def my_decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        ... # Decorator code goes here ...
    return wrapper
```

The usage of `functools.wraps` will ensure the metadata is passed on to the wrapper. If you reapply the decorator `my_decorator` to a fresh function `add` you will see the result looks good:

```
print(add)  # <function add at 0×102db19b0>
```

And using `help(add)` reveals:

```
Help on function add in module __main__:

add(a, b)
    Performs addition!
 Help on add line 1/4 (END) (press h for help or q to quit)
```

Further reading:

- Decorators article.

91 – Timestamp file names

You can combine `datetime.datetime.now` with string formatting to add a timestamp to a file name. This is very helpful to create unique file names dynamically.

Here is an example function using this idea:

```
import datetime

def make_file_name(prefix, extension):
    ts = f"{datetime.datetime.now():%Y%m%d%H%M%S}"
    return f"{prefix}{ts}{extension}"

print(make_file_name("screenshot_", ".png"))
# screenshot_20250707142204.png
```

The `%` specifiers determine what parts of the current date and time make it to the timestamp and this example uses six:

Specifier	Meaning
`%Y`	year
`%m`	month
`%d`	day
`%H`	hour
`%M`	minute
`%S`	second

All but `%Y` result in a 2-digit number which might be 0-padded.

For most applications, going down to the minutes or seconds is enough. If you need microseconds, you can add the specifier `%f`.

92 – Temporary directories

Use the module `tempfile` from the standard library to create a temporary directory. The context manager `TemporaryDirectory` cleans up the directory and its contents when closed.

```
import tempfile

with tempfile.TemporaryDirectory() as tmpdirname:
    print(f"Created temp directory {tmpdirname}.")
    ...  # Do whatever you want inside this directory ...
```

Once the context manager is closed, everything is cleaned up – including the contents of the directory – so you can use it to create and manipulate other files that will be automatically cleaned up.

For example, I used this to edit a video: my program creates a temporary directory and then copies the excerpts of the video that I care about into small segments that are saved in the temporary directory. Then, the segments are all pasted together into a single video that is saved elsewhere. Once the temporary directory goes away, the small segments are also cleaned up.

93 – Non-local variables

To access non-local variables in closures, use the keyword `nonlocal`. Non-local variables are variables that are neither local, nor global. For example, the variable `counter` from the point of view of the function `inner` in the snippet below:

```
def outer():
    counter = 0

    def inner():
        ...  # From the point of view of `inner`,
        # `counter` is neither local nor global.
```

Non-local variables can be read from inner functions without any special keywords:

```
def outer():
    counter = 0

    def inner():
        print(counter)

    return inner

inner = outer()
inner()  # 0
inner()  # 0
```

To write to them, you need to declare the variable as non-local with the keyword `nonlocal`:

```
def outer():
    counter = 0

    def inner():
        nonlocal counter
        counter += 1
        print(counter)

    return inner

inner = outer()
inner()  # 1
inner()  # 2
```

By the way, since you can read the value of a variable without using the keyword `nonlocal`, you can also mutate the value of a variable without the keyword `nonlocal`:

```
def outer():
    my_list = []

    def inner():
        my_list.append(1)
        print(my_list)

    return inner
```

(continues on next page)

(continued from previous page)

```
inner = outer()
inner()  # [1]
inner()  # [1, 1]
```

94 – Dynamic width string formatting

String formatting accepts formatting specifiers dynamically if you specify them within an extra set of curly braces `{}`. This is useful, for example, to compute the maximum width of a column from a list of strings to format them neatly:

```
fruits = ["banana", "cantalope", "pear"]
max_width = max(map(len, fruits))
# max_width is 9

for idx, fruit in enumerate(fruits, start=1):
    print(f"{fruit:<{max_width}} - {idx}")
#                  ^^^^^^^^^^^
```

```
banana    - 1
cantalope - 2
pear      - 3
```

Note how all fruits are left aligned on a field that is as wide as the word "cantalope", since the variable `max_width` has the value `9`. To tell the string formatting that the width is supposed to be the value of the variable `max_width`, you specify the width as `{max_width}` *inside* the format.

95 – Docstring `__doc__` attribute

Docstrings are saved as an attribute `__doc__` on the objects they were defined in. (Both functions and classes.)

This `__doc__` is a string attribute that is both readable and writable. This is used by the built-in `help`, by `functools.wraps` in decorators, and more.

As an example, consider the function `fn` with a short docstring:

```
def fn():
    """Docstring!"""
```

If you use the built-in `help`, you'll see the docstring:

```
help(fn)
"""
fn()
    Docstring!
"""
```

You can read and write to this attribute:

```
print(fn.__doc__)  # Docstring!
fn.__doc__ = "Bye!"
```

Then, things like the built-in `help` will see the new value for the docstring:

```
help(fn)
"""
fn()
    Bye!
"""
```

96 – Common `__hash__` implementation

The dunder method can be implemented to make your instances hashable. This will let you use your instances of your custom class in dictionaries, sets, caches, and other situations.

The most common approach to implement a reasonable `__hash__` is to create a tuple with all the "important attributes" of your instance and then using the hash of that tuple.

As a rule of thumb, these "important attributes" tend to be the ones you would use to determine if two instaces of your class are equal. For example, consider the mockup class `Fraction`:

```
class Fraction:
    def __init__(self, numerator, denominator):
        self._num = numerator
        self._den = denominator
        self._is_simplified = ...
```

The attribute `_is_simplified` will flag whether the fraction has been simplified to its lowest terms, like in `Fraction(1, 3)`, or not, like in `Fraction(3, 9)`. (Note that `1/3` and `3/9` are the same number.)

The fractions `Fraction(1, 3)` and `Fraction(3, 9)` are equal, although their attribute `_is_simplified` is different, since you can use the attributes `_num` and `_den` to determine they have the same numerical value.

Hence, for the class `Fraction`, you would use the attributes `_num` and `_den` to implement `__hash__`:

```
class Fraction:
    # ...

    def __hash__(self):
        return hash((self._num, self._den))
```

Note that, if two instances are considered equal, then their hashes *must be the same*.

97 – Match word boundaries

The special character `\b` can be used to match word boundaries in regular expressions. The special character `\B`, on the other hand, only matches inside words.

By combining `\b` and `\B` you can match prefixes, suffixes, standalone words, and more.

As an example, consider the sentence "watermelon is 92% water!" and three different patterns:

1. The pattern `r"water"` matches the five characteres "water" in sequence, so it would match the two occurrences of "water" in the sentence given.
2. The pattern `r"\bwater\b"` matches "water" as a standalone word, so it would only match the word at the end of the string.
3. The pattern `r"\bwater\B"` matches "water" as a prefix, so it would only match the string "water" that is a substring of the word "watermelon" at the beginning of the sentence.

98 – Custom containment checks

The dunder method `__contains__` can be used to implement custom containment checks in your classes. This will allow instances of your classes to be used with the keywords `in` and `not in`.

As an example, consider the class `ClosedInterval` shown below, that represents all numbers between the left and right boundaries, inclusive:

```
class ClosedInterval:
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def __contains__(self, value):
        return self.left <= value <= self.right
```

By defining the dunder method `__contains__`, instances of `ClosedInterval` can be used with `in` and `not in`:

```
interval = ClosedInterval(2.7, 4.5)

print(3 in interval)  # True
print(7 in interval)  # False
print(7 not in interval)  # True
```

The dunder method `__contains__` accepts the value that is being checked as its only argument and must return a Boolean value that must be `True` if the argument is contained in the instance and `False` if not.

99 – Readable object names

Functions and classes have a dunder attribute `__name__` that is a string with the readable/user-defined name of that object:

```
class Example:
    pass

print(Example.__name__)  # Example
print(int.__name__)  # int
```

```
def foo():
    pass

print(foo.__name__)  # foo
print(print.__name__)  # print
```

This is useful, for example, when defining a class's `__repr__`:

```
class Point:
    def __init__(self, x):
        self.x = x

    def __repr__(self):
        return f"{type(self).__name__}({self.x})"
```

By using the dunder attribute `__name__`, the implementation of `__repr__` will remain correct if the name of the class changes and it is also more likely to stay correct if `Point` is subclassed.

100 – Filtering Truthy values

The built-in `filter` accepts the special value `None` as its first argument, instead of a predicate function. When `None` is the first argument, `filter` will remove all Falsy values from the iterable passed as the second argument.

```
words = ["This", "", None, "list", "contains", "", "some", "words"]

for word in filter(None, words):
    print(f"Word {word!r} has length {len(word)}.")
```

```
Word 'This' has length 4.
Word 'list' has length 4.
Word 'contains' has length 8.
Word 'some' has length 4.
Word 'words' has length 5.
```

In the snippet above, by using `filter(None, ...)`, the loop skips over the empty strings `""` in the list `words` and also the value `None`, since those are all Falsy.

101 – Pretty-printing nested data structures

The module `pprint` from the standard library provides pretty-printing functionality. The most straightforward way of using it is through the function `pprint.pprint`, which pretty-prints its arguments. It is particularly useful when printing nested data structures:

```
>>> import pprint
>>> pprint.pprint(globals())
{'__builtins__': <module 'builtins' (built-in)>,
 '__doc__': None,
 '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0×10324d910>,
 '__name__': '__main__',
 '__package__': None,
 '__spec__': None,
 'partial': <class 'functools.partial'>,
 'pprint': <function pprint at 0×10362c1a0>}
```

102 – methods `__str__` and `__repr__`

Objects can use two different dunder methods to create string representations for themselves:

- `__str__`
- `__repr__`

`__str__` is supposed to provide a "pretty" string representation and `__repr__` is supposed to provide a "debug" string representation.

If `Point` is a class used to represent a point in 3D space, here's what its implementations of `__str__` and `__repr__` could be:

```
class Point:
    def __init__(self, x, y, z):
        self.x, self.y, self.z = x, y, z

    def __str__(self):
        return f"({self.x}, {self.y}, {self.z})"

    def __repr__(self):
        return f"Point({self.x}, {self.y}, {self.z})"
```

When using the built-in `print`, the pretty string representation (`__str__`) is used:

```
print(Point(1, 2, 3))  # (1, 2, 3)
```

When using the built-in `repr`, the formatter `!r` inside f-strings, or when instances of the class `Point` are inside other containers, the debugging string representation (`__repr__`) is used:

```
some_points = [Point(0, 0, 0), Point(1, 2, 3)]
print(some_points)  # [Point(0, 0, 0), Point(1, 2, 3)]
```

As a good rule of thumb, you should be able to copy & paste the output of `__repr__` to recreate the object. Also, when in doubt, implement `__repr__`. I personally rarely implement `__str__`, and when `__repr__` is present, `__str__` falls back to `__repr__`.

Further reading:

- `str` and `repr`

103 – Typing `*args` and `**kwargs`

When typing functions with an arbitrary number of positional arguments (`*args`) or with an arbitrary number of keyword arguments (`**kwargs`), the type hints should target the values that you'll accept. In other words, don't add type hints for the tuple `args` as a whole or for the dictionary `kwargs` as a whole.

The signature below denotes a function `my_max` that accepts a variable number of integers and a variable number of keyword arguments that are Booleans:

```
def my_max(
    *args: int,
    **kwargs: bool,
) -> int:
    ...
```

On the other hand, the signature below denotes a function `my_max` that accepts a variable number of tuples of integers and a variable number of keyword arguments that are dictionaries that map strings to Booleans:

```
def my_max(
    *args: tuple[int, ...],
    **kwargs: dict[str, bool],
) -> int:
    ...
```

104 – AST parsing

The module `ast` has a function `parse` that you can use to parse code (in a string) into what's called an AST. An AST is an "Abstract Syntax Tree", an intermediate representation of your code that Python operates on.

Studying the ASTs produced by the module `ast` can teach you a lot about the Python language. For example, by using `ast.parse` you can find out that the keyword `elif` is just syntactic sugar for a sequence of `else: if ...:` at the language level:

```
import ast

code = """\
if a:
    pass
else:         # <--
    if b:     # <--
        pass
    else:
        pass
"""

print(ast.dump(
    ast.parse(code),
    indent=2,
))
```

```
Module(
  body=[
    If(
      test=Name(id='a', ctx=Load()),
      body=[
        Pass()],
      orelse=[                              <--
        If(                                 <--
          test=Name(id='b', ctx=Load()),
          body=[
            Pass()],
          orelse=[
            Pass()])])])
```

This structure is **identical** to the one you get if you use `elif` instead of `else: if ...:`:

```
import ast

code = """\
if a:
    pass
elif b:   # <--
    pass
else:
    pass
"""
```

(continues on next page)

(continued from previous page)

```
print(ast.dump(
    ast.parse(code),
    indent=2,
))  # Exact same output.
```

105 – Dunder method `__missing__`

The dunder method `__missing__` is part of the `dict` data model. If you look up a key that isn't in the dictionary, `__missing__` is called to let you handle the missing key.

```
class Test(dict):
    def __missing__(self, key):
        print(f"In __missing__ with {key = }.")

d = Test()
d[42]  # Output: In __missing__ with key = 42.
```

This can be used, for example, to implement `collections.defaultdict` yourself:

```
class defaultdict_(dict):
    def __init__(self, factory, **kwargs):
        super().__init__(**kwargs)
        self.default_factory = factory

    def __missing__(self, key):
        """Populate the missing key and
        return its value."""
        self[key] = self.default_factory()
        return self[key]

# int() gives 0.
olympic_medals = defaultdict_(int)
olympic_medals["Phelps"] = 28

print(olympic_medals["Phelps"])  # 28
print(olympic_medals["me"])  # 0 :(
```

Further reading:

- Module `collections` overview

106 – Custom `enum` search behaviour

Similar to the dunder method `__missing__`, enumerations can have a class method `_missing_` (with a single underscore).

This class method is called when instantiating an enumeration with a value that's not legal. The class method can implement custom search behaviour (e.g., error-tolerant search) to try and find/return the correct member.

As a silly example, the enum shown below uses a naive strategy to try and suppress a common difference between UK and US English spelling:

```
from enum import auto, StrEnum

class Proximity(StrEnum):
    NEIGHBOR = auto()
    DISTANT = auto()

    @classmethod
    def _missing_(cls, value):
        try_value = value.casefold().replace("ou", "o")
        for member in cls:
            if member.value.casefold() == try_value:
                return member
        return None
```

Here's `_missing_` in action:

```
#                       vv
print(Proximity("NEIGHBOUR"))
# <Proximity.NEIGHBOR: 'neighbor'>
#                   ^          ^
```

107 – `map`'s keyword argument `strict`

Akin to `zip`'s keyword argument `strict`, in Python 3.14 the built-in `map` got a keyword argument `strict`. By setting `strict=True`, you get a `ValueError` if the iterables have different lengths:

```
bases = [2, 3, 4, 2, 3, 4]
exps = [2, 2, 2]

for num in map(pow, bases, exps, strict=True):
    print(num, end=" ")
# 4 9 16
# ValueError
```

In 99% of the situations where you use `map` *with multiple arguments* you'll want to set this, so don't forget it!

108 – Dynamic module attribute look-up

Modules can implement the dunder method `__getattr__`, which can be used to dynamically load names into a module. This can be useful to create lazy imports or to issue warnings when certain things are imported.

Suppose `HeavyClass` is available from the module `my_module`, but suppose that `HeavyClass` takes a while to initialise/import and therefore you want to be able to import it lazily. Then, you'd define this `__getattr__` in `my_module`:

```
def __getattr__(name):
    if name == "HeavyClass":
        print("Lazy importing ... ")
        from _module import HeavyClass
        globals()[name] = HeavyClass
        return HeavyClass

    raise AttributeError(f"module {__name__!r} has no attribute {name!r}")
```

Then, you can use `HeavyClass` elsewhere. The first time you use it, you'll trigger `__getattr__`, but the second time you won't because the lazy import is then saved in the global namespace of the module.

```
import my_module

obj1 = my_module.HeavyClass()
# Lazy importing ...

obj2 = my_module.HeavyClass()
```

This is used in the modules `typing` and `collections`, for example!

109 – Add typing to decorators

Decorators used to be nasty to type. How do you create an inner, general function that you then type in a way that depends on the function(s) you will be decorating?!

Python fixes this with `typing.ParamSpec` (added in Python 3.10), which allows you to refer to the signature of your original function in the signature of your new function.

In general, a decorator will look like this:

```
def decorator(f):  # This decorator does nothing in practice.
    def wrapper(*args, **kwargs):
        return f(*args, **kwargs)
    return wrapper
```

Given this decorator, the diagram below shows a tiny usage example with the 3.12+ type parameter syntax:

```
from typing import Callable

def decorator[T, **P](f: Callable[P, T]) -> Callable[P, T]:
    def wrapper(*args: P.args, **kwargs: P.kwargs):
        return f(*args, **kwargs)

    return wrapper
```

The `[..., **P]` syntax creates a parameter specification (`typing.ParamSpec`) called `P`. Then, the signature `(f: Callable[P, T]) -> Callable[P, T]` means that the decorator accepts a function whose **param**eter **spec**ification is `P`, and returns a function with the same parameter specification. In other words, the decorator will preserve the signature of the function. More specifically, the `.args` and `.kwargs` attributes of the `ParamSpec` link the original types of the arguments of `f` to the types of the arguments of `wrapper`.

110 – Non-greedy regex quantifiers

The quantifiers `*`, `+`, and `?`, in regular expressions, are all greedy. This means that they will try to match as much as possible, while still allowing the full pattern to match:

```
import re

html = "<a href='mathspp.com'>My site</a>"
#                            ^          ^
# Note how there are two '>' in the string.

greedy = re.compile(r"<a.*>")

print(re.match(greedy, html).group())
# <a href='mathspp.com'>My site</a>
```

However, sometimes you don't want to match as much as possible; you want to match as little as possible. When that's the case, you can add a question mark `?` to the quantifiers, making them `*?`, `+?`, and `??`. These are non-greedy and will try to match as few characters as possible:

```
non_greedy = re.compile(r"<a.*?>")

print(re.match(non_greedy, html).group())
# <a href='mathspp.com'>
```

111 – Type hints that refer to functions

When dealing with higher-order functions (functions that accept functions as arguments or that return functions), you can use `collections.abc.Callable` to add a type hint to the function that is an argument/the return value.

`Callable` takes two values:

1. the list of types of the arguments of the callable; and
2. the return type of the callable.

For example, `Callable[[str, bool], None]` is the type that refers to functions that accept a string and a Boolean value and return `None`.

112 – Two-dimensional range

You can build a two-dimensional range using the built-in `range` and `itertools.product`. This is useful when traversing 2D structures like grids.

It's also better than a nested for loop because it's easier to break out of:

```
def range_2d(n, m):
    return product(range(n), range(m))

for x, y in range_2d(10, 3):
    if x == 2:
        break
    print(x, y)
"""Output:
0 0
0 1
0 2
1 0
1 1
1 2
"""
```

113 – Constant variables

There is no way to create a true constant in Python. Instead, you can use other mechanisms to signal to people that some values shouldn't change.

PEP 8 establishes the convention that if a variable has its name written in `ALL_UPPERCASE`, then that's a constant and you shouldn't modify its value:

```
PI = 3.1415
```

Additionally, the module `typing` has the annotation `Final` that you can use to indicate to type checkers that a given name shouldn't be reassigned to. `Final` is a generic that takes the true type of the variable inside square brackets `[]`:

```
PI: Final[float] = 3.1415
```

114 – `dict.fromkeys`

If you have an iterable of keys you can initialise a dictionary by using the class method `dict.fromkeys`, which creates a dictionary where all keys map to the value `None`:

```
keys = ["name", "age", "address"]
person_info = dict.fromkeys(keys)

print(person_info)
```

```
{
    'name': None,
    'age': None,
    'address': None,
}
```

If the default value `None` doesn't suit you, you can change it by passing a different default value as the second argument to `dict.fromkeys`:

```
person_info = dict.fromkeys(keys, "")

print(person_info)
```

```
{
    'name': "",
    'age': "",
    'address': "",
}
```

Be careful when passing mutable values as the default value, though:

```
keys = ["a", "b", "c"]
my_dict = dict.fromkeys(keys, [])

my_dict["a"].append("Hello")
my_dict["b"].append("there")
print(my_dict["c"])  # ['Hello', 'there']
```

Further reading:

- Custom value in `dict.fromkeys`.

115 – Double leading underscore

Name mangling is a Python feature that mangles the names of attributes and methods of functions that start with a leading double underscore, like `__value`.

Attributes/methods like this are accessible with their original name from inside the class:

```
class MyClass:
    def __init__(self):
        self.__value = 42

    def print_value(self):
        print(self.__value)

c = MyClass()
c.print_value()  # 42
```

From outside the class, attributes/methods have their names mangled with a leading underscore `_` and the name of the class:

```
print(c.__value)  # AttributeError
#       vvvvvvvv mangled
print(c._MyClass__value)  # 42
```

This feature is not supposed to be used to create "private" attributes, since the attributes are still public. This is intended in OOP contexts like when writing mix-ins, to make it virtually impossible for attributes/methods to be overridden by mistake.

116 – Send data into generators

Generators have a method `send` that you can use to send data into the generator. Inside the generator, that data comes from the expression `yield`, that evaluates to the value that was sent into the generator.

The example below shows a generator that produces consecutive integers. However, if you send an integer into the generator, you'll skip ahead that many integers:

```
def count():
    at = 0
    while True:
        skip = yield at
        if skip is not None:
            at += skip
        else:
            at += 1
```

Here's an example usage:

```
counter = count()
print(next(counter))  # 0
print(next(counter))  # 1

print(counter.send(5))  # 6

print(next(counter))  # 7
```

Note how the return value of the method `send` is the next value produced by the generator.

When you use the built-in `next` to fetch the next value the result of the `yield` is None, because no value was sent to the generator.

117 – Dictionary creation idiom

Dictionaries can be created from iterables of key/value pairs, which naturally gives rise to a useful idiom to create dictionaries using the built-ins `zip` and `dict`.

Instead of iterating over two iterables, each containing all keys or all values:

```
my_dict = {}
for key, value in zip(keys, values):
    my_dict[key] = value
```

Or even instead of using a dictionary comprehension:

```
my_dict = {key: value for key, value in zip(keys, values)}
```

You can use the built-ins `dict` and `zip`:

```
my_dict = dict(zip(keys, values))
```

Here's a concrete example:

```
prompts = ["age", "country", "favourite language"]
answers = [28, "Portugal", "Python"]

info = dict(zip(prompts, answers))
print(info)
```

```
{
    'age': 28,
    'country': 'Portugal',
    'favourite language': 'Python',
}
```

118 – Safely overriding methods

The decorator `typing.overrides` can be used to mark that a method is overriding a method from a parent class. If it isn't, type checkers will complain.

Here's an example super class that implements two methods `sound` and `sleep`:

```
class Animal:
    def sound(self) -> str:
        ...

    def sleep(self) -> None:
        ...
```

Suppose you want to override this class to create a class `Cat`:

```
from typing import overrides

class Cat(Animal):
    @overrides
    def sound(self) -> str:
        ...

    @overrides
    def seep(self) -> None
        ...
```

A type checker will see that `Cat.sound` and `Cat.seep` are supposed to be overrides of methods from the parent class(es). Since a type checker won't find a method `seep` in a parent class (because it's supposed to be "`sleep`", with an extra L), you will get an error and you'll easily spot your typo.

Using `overrides` is also useful if you're inheriting from parent classes from other packages, since it will let you know if a method name changes; for example, if the API changes in a version upgrade.

119 – Efficiently extending a list

The method `extend` is the efficient way of adding multiple elements to a list in one go:

```
my_list = [...]

my_list.extend(other_list)
```

This also works with other iterables, even if they're not lists.

A common alternative that also works with non-list iterables is using a `for` loop that calls the method `append` repeatedly:

```
my_list = [...]

for value in other_list:
    my_list.append(value)
```

However, this is inefficient because the repeated calls to `append` will waste time constantly resizing the list.

Another alternative that may sound like a good idea but actually wastes time and memory computing a third list is concatenating the two lists:

```
my_list = [...]

my_list = my_list + other_list
```

CPython implementation detail: if you use augmented assignment `+=` then it's efficient again because it defers to the method `extend` at the C level.

120 – All equal

To check if all values of an iterable are equal (up to a `key` function), you can use `itertools.groupby` and check how many groups are found (under the same `key` function):

```
from itertools import groupby, islice

def take(iterable, n):
    return list(islice(iterable, n))

def all_equal(iterable, *, key=None):
    return len(take(
        groupby(iterable, key),
        2,
    )) <= 1
```

The function `all_equal` uses `groupby` to create runs of elements that are the same under the function `key`. Then, the function `take` is used to try and take the first 2 of those runs. If there are no runs, or if there is only one run, it's because all values were the same.

121 – Paginate results

You can use `itertools.batched` to paginate an arbitrary iterable. For example, you can take a generator that produces single values and turn it into a generator that produces "pages" of results.

You can even do this with an auxiliary generator:

```
def paginated(function):
    def wrapper(*args, **kwargs):
        yield from batched(function(*args, **kwargs), 10)
    return wrapper
```

Here's an example application:

```
@paginated
def evens():
    at = 0
    while True:
        yield at
        at += 2

for page in evens():
    print(page)
# (0, 2, 4, 6, 8, 10, 12, 14, 16, 18)
# (20, 22, 24, 26, 28, 30, 32, 34, 36, 38)
# ...
```

122 – Ignore exceptions

The module `contextlib` provides a context manager `suppress` that you can use to suppress (ignore) a given exception. Very helpful when you want to run some code, hoping it works, but when you really don't care if it ends up failing. You can read this idiom exactly as "it's ok if this fails":

```
from contextlib import suppress

with suppress(SomeException):
    run_some_code()
```

This is better than using `except SomeException: pass` because suppress conveys the meaning "ignore the exception" immediately, whereas using `try: ... except: pass` only shows that information much later:

```
try:
    run_some_code()
except SomeException:
    pass
```

Further reading:

- Ignoring exceptions with `contextlib.suppress`

123 – Using properties

The built-in decorator `property` can be used to turn a method into an attribute. This means that properties look like attributes but actually run methods that compute the values that must be returned.

Useful, for example, if you have something that "looks" like an attribute but needs to be computed dynamically, like someone's age based on their birthdate:

```
import datetime as dt

class Person:
    def __init__(self, name, birth_date):
        self.name = name
        self.birth_date = birth_date

    @property
    def age(self):
        y, m, d, *_ = dt.date.today().timetuple()
        by, bm, bd = self.birth_date
        return y - by - ((bm, bd) > (m, d))


john = Person("John", (1990, 8, 28))

# 27th August, 2025, it's John's birthday tomorrow
print(john.age)  # 34

# time.sleep(1 day)

# 28th August, 2025, it's John's 35th birthday
print(john.age)  # 35
```

Note how `Person.age` is implemented as a method, but because of the decorator `property`, it's accessed as an attribute.

Further reading:

- Properties

124 – Operate on two lists of numbers

If you combine the module `operator` with `itertools.starmap` and `itertools.zip_longest`, you can write any function that merges two lists by operating on respective pairs of elements.

`zip_longest` allows you to operate until the end of the longer list and then `starmap` allows you to unpack the tuples into arguments of the operation you chose. If you don't need `zip_longest` to get to the end of the longer iterable, then you can just use the built-in `map(add, l1, l2)` and `starmap` ends up not being needed either.

The example below adds two lists, element by element, padding the shorter list with zeroes:

```
from itertools import starmap, zip_longest
from operator import add

def add_lists(l1, l2):
    """Function to add respective elements from two lists."""
    return list(starmap(add, zip_longest(l1, l2, fillvalue=0)))

add_lists(
    [1, 2, 3],
    [10, 20, 30, 40],
)  # [11, 22, 33, 40]
```

125 – Enforce positional arguments

You can use the forward slash `/` in a function signature to force arguments to be passed in as positional only. In other words, use `/` to disallow arguments from being passed as keyword arguments.

This useful when the names of the arguments are inconsequential. For example, in a mathematics context, `p` and `q` are very common names for polynomials, but the names themselves have no meaning. So, if you're defining a function that adds two polynomials (using *the idiom to operate on two lists of numbers*), you might want `p` and `q` to only be passed in positionally:

```
from itertools import starmap, zip_longest
from operator import add

def add_polynomials(p, q, /):
    """Function to add two polynomials as lists of coefficients."""
    return list(starmap(add, zip_longest(p, q, fillvalue=0)))

add_polynomials(poly1, poly2)  # Ok.
add_polynomials(p=poly1, q=poly2)  # TypeError!
```

Another example use case is from the Cyclopts package, a package to create CLIs. In Cyclopts, you use the forward slash `/` to declare that a parameter is an argument and not an option.

126 – Integer literals in multiple bases

Python integer literals are base 10 by default. But you can also use the prefixes `0b`, `0o`, and `0x`, to write integer literals in binary, octal, and hexadecimal, respectively:

Prefix	Base
`0b`	binary
`0o`	octal
`0x`	hexadecimal

```
feature_flags = 0b1101  # turn off *that* feature

file_permissions = 0o755  # default file permissions

magenta = 0×CE5D97  # accent colour used on my site
```

Further reading:

- Base conversion in Python

127 – Remove punctuation functionally

You *already know how to remove punctuation from a string*, but you can leverage `functools.Placeholder` (new in Python 3.14) and `functools.partial` to turn that into a function:

```
from functools import Placeholder as _P, partial
import string

remove_punctuation = partial(
    str.translate,
    _P,
    str.maketrans("", "", string.punctuation),
)
```

Now, you can pass a string into the function `remove_punctuation`, which puts it as the first argument to `str.translate` because of the usage of `Placeholder`:

```
print(remove_punctuation("Hello, world!"))
# Hello world
```

This wouldn't have been possible without `Placeholder` because `str.translate` only accepts positional-only arguments.

Further reading:

- `functools.Placeholder`
- `functools.partial`
- String `translate` and `maketrans` methods

128 – `pairwise` generalisation

`itertools.pairwise` accepts an iterable and produces overlapping pairs of consecutive elements:

```
from itertools import pairwise

my_list = [42, 73, 16, 0, 10]
for a, b in pairwise(my_list):
    print(a, b)

"""
Output:
42 73
73 16
16 0
0 10
"""
```

If you need a similar behaviour but with tuples of an arbitrary size, you can implement that as a generator with `collections.deque` and `itertools.islice`:

```
from collections import deque
from itertools import islice

def n_tuples(data, n):
    """Produces overlapping tuples of size `n`."""
    data = iter(data)
    window = deque(islice(data, n - 1), maxlen=n)
    for value in data:
        window.append(value)
        yield tuple(window)
```

129 – Prevent subclassing/overriding

The decorator `typing.final` (lowercase **f**) can be used to mark classes that shouldn't be subclassed:

```
from typing import final

@final
class Base:
    pass

class C(Base):  # Type checkers will complain.
    pass
```

The decorator can also be used to mark methods that shouldn't be overridden by subclasses:

```
from typing import final

class Base:
    @final
    def foo(self) -> None:
        pass

class C(Base):  # This is fine ...
    def foo(self) -> None:  # But a type checker will complain here.
        pass
```

130 – Make numbers more readable

You can use the character underscore _ in number literals (integers, floats, and complex numbers), to separate groups of digits. This works for *any base that Python supports*:

- grouping digits by thousands in base 10;
- grouping bits by 4 or 8 bits in base 2; and
- grouping RGB channels in colours represented in hexadecimal,

are just some examples:

```
x = 12_345_678  # integer
almost_x = 12_345_677.99  # float
complex_x = 1+23_456j  # complex

binary = 0b1110_0100
octal = 0o7351_2242
hexadecimal = 0×ff_d3_ab
```

131 – Efficiently count words in a string

You can efficiently count how many words a string has by using `itertools.groupby`:

```
from itertools import groupby

def count_words(s):
    return sum(is_alpha for is_alpha, _ in groupby(s, key=str.isalpha))
```

This is so efficient, you could use it to count words on files larger than your computer's RAM memory. But here's an example usage with a smaller string:

```
print(
    count_words("Hey there—how you doin'?")
)  # 5
```

Note that, depending on the predicate you use as the keyword argument `key`, you get different definitions of "word". By using `is_alpha`, you are implicitly defining a "word" to be a sequence of characters for which the string method `str.isalpha` returns `True`.

Under the hood, `groupby` is grouping characters in runs of characters for which `str.isalpha` is `True` and runs for which it is `False`, and then you're counting the number of runs associated with the value `True`.

132 – Efficiently count characters in a string

Python has a specialised dictionary for whenever you need to count things or represent counts: `collections.Counter`. The only restriction you need to keep in mind is that `Counter` can only count hashable objects, since it's dictionary-like.

You can initialise by passing in an iterable of elements to count:

```
from collections import Counter

counts = Counter("mississippi")
```

From there, you can access counts by key, even if the key isn't there:

```
print(counts["m"])  # 1
print(counts["z"])  # 0
```

Since it's a specialised dictionary, `Counter` also includes many useful methods to work with counts, like the method `most_common`:

```
print(counts.most_common(2))
# [('i', 4), ('s', 4)]
```

133 – Find files in a directory

You can use `pathlib.Path` together with the methods `iterdir` and `is_file` to find and list all files inside a directory:

```
from pathlib import Path

directory = Path("path/to/dir")
files = [p for p in directory.iterdir() if p.is_file()]
```

The method `iterdir` finds all directory contents but won't recurse into subdirectories and then `is_file` filters out anything that's not a file. You could replace it with `is_dir` to get all directories instead.

This is more robust that using something like `glob(*.*)` because files don't always have dots in their names.

134 – Invertible flags

The special values `True` and `False` are tightly connected: they mean the opposite of each other and you can negate one to get the other:

- `not True` gives `False`
- `not False` gives `True`

You can create the same behaviour and the same semantics of "two opposite values" by using a flag enumeration with only two members with values `0` and `1`:

```
from enum import Flag

class DoorState(Flag):
    CLOSED = 0
    OPEN = 1

print(~DoorState.CLOSED)  # DoorState.OPEN
print(~DoorState.OPEN)    # DoorState.CLOSED
```

The bitwise operator `~` (NOT) negates the flag, turning it into the opposite member.

You can also use the keyword `not` to do the negation, but in that case you need to reinstantiate the flag to make it a member of the enumeration again:

```
print(DoorState(not DoorState.CLOSED))
# DoorState.OPEN
print(DoorState(not DoorState.OPEN))
# DoorState.CLOSED
```

Useful for user configurations where you want readable options!

135 – Class and instance attributes

In a typed project, you can specify the types of (instance) attributes by declaring them all in the body of the class (just like dataclasses do):

```
class Person:
    name: str
    ...

p = Person()
p.name = "Steve"  # Type checks.
p.name = 73       # Type error.
```

But if you want to declare a class attribute (one that shouldn't be modified from any instance) then you must use `typing.ClassVar` to distinguish from a normal instance attribute:

```
from typing import ClassVar

class Person:
    species: ClassVar[str] = "Homo sp"
    name: str
    ...

Person.species = "Homo sapiens"  # Type checks.
p = Person()
p.species = "..."  # Can't modify class attribute from instance.
p.name = "Steve"  # Still an instance attribute.
```

136 – Strict batching

When using `itertools.batched` with the purpose of reshaping a one-dimensional iterable into sections of a fixed size, set `strict=True` if you expect the iterable to contain a whole number of sections.

Without it, the final batch may be incomplete:

```
from itertools import batched

for batch in batched(range(10), 4):
    print(batch)
# (0, 1, 2, 3)
# (4, 5, 6, 7)
# (8, 9)
```

If you set `strict=True`, you get a `ValueError` if the final batch is smaller than the specified size:

```
from itertools import batched

for batch in batched(range(10), 4):
    print(batch)
# (0, 1, 2, 3)
# (4, 5, 6, 7)
# ValueError
```

Note that you only get the `ValueError` once you get to the end of the batched iterable and *not* when you create the batched iterable:

```
batched(range(10), 4)  # No error raised yet.
```

137 – Read-only attributes

The standard way of creating read-only attributes in Python is by creating a "private" attribute and then exposing it through a property:

```
class Person:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name
```

This way, the property is read-only:

```
john = Person("John")

print(john.name)  # John
john.name = "Steve"  # AttributeError
```

However, the underlying "private" attribute is still accessible and could be modified:

```
john._name = "Steve"
print(john.name)  # Steve
```

138 – Alternative constructors as class methods

A common use case for class methods is to provide alternative constructors for classes. (*An example from the built-ins is* `dict.fromkeys`.)

A class method accepts the class as its first argument, not `self`. (Otherwise, it would be an instance method and not a class method.)

```
class Person:
    def __init__(self, first, last):
        self.first = first
        self.last = last

    @classmethod
    def from_name(cls, name):
        return cls(*name.split())
```

The class method `from_name` lets you create instances of the class `Person` by passing the full name directly:

```
john = Person.from_name("John Smith")

print(type(john))  # <__main__.Person object at 0×103a26e40>
print(john.first)  # John
print(john.last)  # Smith
```

139 – Reduce boilerplate with dataclasses

The module `dataclasses` provides a decorator `dataclass` that you're supposed to use on classes. The decorator `dataclass` reduces the need to write boilerplate code by adding the dunder methods `__init__`, `__eq__`, and `__repr__`, to the class that is decorated.

To create a dataclass, all you need is to define the attributes that are relevant by listing them in the class body along with an annotation of their type:

```
from dataclasses import dataclass

@dataclass
class Person:
    first: str
    last : str
```

The four lines of code that define the class `Person` as a dataclass are roughly equivalent to the following handwritten definition:

```
class Person:
    def __init__(self, first, last):
        self.first = first
        self.last = last

    def __eq__(self, other):
        if isinstance(other, Person):
            return self.first == other.first and self.last == other.last
        return NotImplemented

    def __repr__(self):
        cls = type(self).__name__
        return f"{cls}(first={self.first!r}, last={self.last!r})"
```

140 – Compute partial sums with `accumulate`

`itertools.accumulate` computes partial sums of the given iterable. Or, in other words, `accumulate` computes prefix sums.

For example, if you give it a list of bank movements, `accumulate` computes the intermediate bank balances:

```
from itertools import accumulate

movements = [100, -200, 300, -450]
initial_balance = 1000

for balance in accumulate(movements, initial=initial_balance):
    print(balance)
# 1000
# 1100 (1000 + 100)
# 900  (1100 - 200)
# 1200 (900 + 300)
# 750  (1200 - 450)
```

This shows there is a relationship between `accumulate` and the built-in `sum` if you consider slices of the list `movements` with the initial balance at the front:

```
    [1000, 100, -200, 300, -450]
sum(     )                          # 1000
sum(          )                     # 1100
sum(               )                # 900
sum(                    )           # 1200
sum(                         )      # 750
```

141 – Regex matches across newlines

The special character `.` (dot) matches anything in the context of a regular expression, except for the newline character:

```
import re

pattern = r"<!--.*⟶"  # HTML multiline comment.
text = """# Python drops <!--This
is a multiline comment.⟶"""
```

```
print(re.search(pattern, text))  # None
```

If you want to use the special character `.` (dot) for matches that potentially span across multiple lines, you'll need to use flag `re.DOTALL`:

```
print(re.search(pattern, text, flags=re.DOTALL))
# <re.Match object; ...
```

142 – Using fractions

Floats are a bit annoying when you have to do computations because of the inaccuracies inherent to their representation.

For example, `49 * (1 / 49)` should be `1.0`, but Python gives you a different result:

```
print(49 * (1 / 49))  # 0.9999999999999999
```

(In all fairness, most programming languages agree with Python here.)

In situations like this, if precision is important, you can use `fractions.Fraction` to perform 100% accurate computations and therefore having no errors or inaccuracies in your results:

```
from fractions import Fraction

print(49 * Fraction(1, 49))  # 1
```

Note, however, that arithmetic operations performed between fractions or integers and fractions return other fractions:

```
f = 43 + Fraction(1, 49)
print(f)  # 2108/49
print(repr(f))  # Fraction(2108, 49)
```

But operations between fractions and floats produce floats:

```
print(49.0 * Fraction(1, 49))  # 0.9999999999999999
```

143 – Decimals

Fractions are very useful but sometimes it is not very helpful to compute `1 / 7` and get `1/7` as a result…

Instead, you may want to use the module `decimal` and its class `decimal.Decimal` to compute fast, correctly-rounded floating-point arithmetic:

```
from decimal import Decimal

print(1.1 + 2.2)  # 3.3000000000000003

d = Decimal("1.1") + Decimal("2.2")
print(d)  # 3.3
print(repr(d))  # Decimal('3.3')
```

One good thing about the module `decimal` is that you can set the precision (number of decimal places) with which results are computed:

```
from decimal import getcontext

getcontext().prec = 6
print(Decimal(1) / Decimal(7))  # 0.142857

getcontext().prec = 28
print(Decimal(1) / Decimal(7))  # 0.1428571428571428571428571429
```

144 – Safe random tokens

To create a URL-safe token in your web server, use `secrets.token_urlsafe`:

```
from secrets import token_urlsafe

print(token_urlsafe())  # ScIFbN8Nc0NKLI40_8o0cMwkqRRFfVgGUE8kBgEkRf0
```

The function `token_urlsafe` accepts an integer for the number of random bytes to be generated, but that's not the same as the number of characters generated because of the way the random bytes are then turned into a URL-safe string:

```
print(token_urlsafe(4))
# eWtWsw
```

Remember that the module `random` is for modelling/simulations, not for security-sensitive operations or for cryptography!

145 – NotImplemented

The singleton built-in `NotImplemented` has one and only one very specific use case: binary dunder methods return this value when they don't know how to compute the result of the operation between the two given arguments.

In other words, the only correct way of implementing a binary dunder method is to return `NotImplemented` after handling all the cases you know and care about.

Here is an example with a class `Person`:

```
class Person:
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        if isinstance(other, Person):
            return self.name == other.name
        return NotImplemented
```

One might think that the correct thing to do is to return `False` in `Person.__eq__` after concluding that `other` isn't an instance of `Person`, but that precludes you and others from writing classes in the future that can compare to `Person`.

For example, suppose the class `Employee` is implemented at a later date and it's independent of the class `Person`. Employees will only be comparable to persons if `return NotImplemented` is used in `Person.__eq__`:

```
class Employee:
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        if isinstance(other, Person | Employee):
            return self.name == other.name
        return NotImplemented

print(Person("John") == Employee("John"))  # True
```

Give it a try. Replace the final `return` of `Person.__eq__` with `return False` and note how the value that is printed becomes `False` instead of `True`.

`NotImplemented` should not be confused with *the built-in exception* `NotImplementedError`.

146 – Cycling over an iterable

Whenever you need to cycle over an iterable use `itertools.cycle`:

```
from itertools import cycle

servers = [...]

for server in cycle(servers):
    server.ping()
```

This is preferred over the modulo operator trick:

```
servers = [...]
idx = 0

while True:
    idx = (idx + 1) % len(servers)
    servers[idx].ping()
```

The two main advantages are that

1. `cycle` is more readable because the name spells out what you are doing; and
2. the modulo operator trick has limitations, since it only works on iterables that you can index and compute the length of.

147 – String formatting field alignment

There are three main alignment options in string formatting:

| Character | Meaning | | | | | < | align left | | > | align right | | ^ | centre |

```
total = 123

print(f"Total: ${total:<7}.")
# Total: $123    .

print(f"Total: ${total:>7}.")
# Total: $    123.

print(f"Total: ${total:^7}.")
# Total: $  123  .
```

For most objects, the default is to align left:

```
obj = "hi"
print(f"-{obj:10}-")
# -hi        -
```

However, for numbers, the default is to align right:

```
obj = 42
print(f"-{obj:10}-")
# -        42-
```

(There is also a fourth alignment option = that only works with numbers!)

148 – Set operations with operators

Many set operations, like containment checks, can be written with Python's operators instead of method calls. This allows for succinct but expressive code:

```
import string

allowed_chars = set(string.ascii_letters)

s = "I like trains."

if not (used := set(s)) <= allowed_chars:
    print("Illegal characters found:")
    print(used - allowed_chars)

# Illegal characters found:
# {' ', '.'}
```

The comparison `used <= allowed_chars` is the same as `used.issubset(allowed_chars)` and checks whether `used` is contained within `allowed_chars`.

The operation `used - allowed_chars` is the same as `used.difference(allowed_chars)` and computes all elements of `used` that are not in `allowed_chars`.

149 – `NotImplementedError`

The built-in exception `NotImplementedError` is supposed to be used in base classes to indicate that a given method must be implemented by the derived class:

```
class Parser:
    def parse(self, data):
        raise NotImplementedError
```

You can also use the exception `NotImplementedError` to create a stub for a method but also signal that its real implementation still needs to be added:

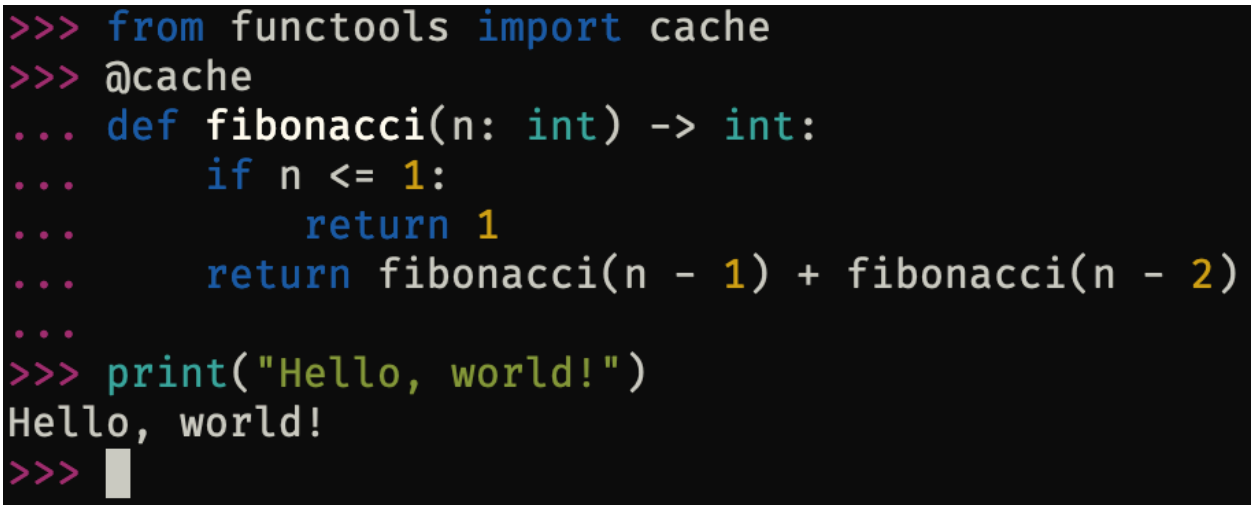
```
class MyParser(Parser):
    def parse(self, data):
        self._validate_data(data)
        # Parse the data ...

    def _validate_data(self, data):
        raise NotImplementedError
```

The documentation also notes that you should **not** use `NotImplementedError` to indicate that a method isn't supposed to be supported at all by a (sub)class.

`NotImplementedError` should not be confused with *the built-in singleton* `NotImplemented`.

150 – Colour in the REPL

Since Python 3.14 that the Python REPL has colours, giving you syntax highlighting directly in the terminal.



CLIs in the standard library like `argparse`, `json`, and `unittest`, also have coloured output now.

151 – copy files

You can copy files and directories to new locations by using `pathlib.Path.copy`.

Suppose your filesystem looks like this:

```
- base/
   - other/
   - myfile.txt
```

If your current working directory is `base`, then the snippet

```
from pathlib import Path

Path("myfile.txt").copy("myfile2.txt")
```

will create a copy of `myfile.txt` named `myfile2.txt` next to it:

```
- base/
   - other/
   - myfile.txt
   - myfile2.txt
```

However, if instead you run the snippet

```
from pathlib import Path

Path("myfile.txt").copy_into("other")
```

then the file `myfile.txt` is copied, with the same name, into the folder `other`:

```
- base/
   - other/
      - myfile.txt
   - myfile.txt
```

The methods `copy` and `copy_into` accept strings or `pathlib.Path` objects, although these two examples used strings.

152 – Find similar words

You can use the function `get_close_matches` from the module `difflib` to find words that are similar to another given word, for example in the context of an auto-correct feature.

Suppose you're trying to write a loop but misspell the keyword `while`:

```
>>> while True:
  File "<python-input-7>", line 1
    while True:
    ^^^^^^
SyntaxError: invalid syntax. Did you mean 'while'?
```

The suggestion of the (correct) keyword `while` is done with `get_close_matches`:

```
from difflib import get_close_matches
import keyword

print(
    get_close_matches("whille", keyword.kwlist)
)  # ['while']
```

The function `get_close_matches` also allows you to specify the maximum number of similar matches you want and the "similarity threshold" to be used.

153 – Forward references in annotations

Forward/cyclic references in Python type hints raise a `NameError` up until Python 3.13:

```
class Node:
    next_node: Node

# NameError: name 'Node' is not defined.
```

You can fix this by turning the type hint `: Node` into a string `: "Node"` or by using a future import:

```
from __future__ import annotations

class Node:
    next_node: Node
```

By using the future import, you are deferring the evaluation of the annotations and that allows you to write cyclic/forward references.

In Python 3.14, this became the default behaviour:

```
# Python 3.14
class Node:
    next_node: Node
# No `NameError`.
```

Further reading:

- PEP 649 – Deferred Evaluation of Annotations Using Descriptors

154 – Compression algorithms

Python 3.14 introduced a new module `compression` that provides a unified interface to access the five compression algorithms that Python supports:

- `bz2`
- `gzip`
- `lzma`
- `zlib`
- `zstd` (new in Python 3.14)

Here's an example usage of `zstd` to compress a sequence of bytes:

```
from compression import zstd

data = "3.141592653589793".encode() * 20
compressed = zstd.compress(data)
ratio = len(compressed) / len(data)
print(ratio)  # 0.1  # <- 10 times smaller!
```

The algorithms `bz2`, `gzip`, `lzma`, and `zlib`, were already standalone modules in Python 3.13 and earlier, so they continue to be so, but you can also access them through the unified interface of the module `compress`.

Further reading:

- Module `compression` overview

155 – `json` CLI

The module `json` is your go-to tool when working with files or data in JSON format. In Python 3.14, the module got a new CLI that allows you to validate and pretty-print JSON data directly from the command line.

For example, if the file `mydata.json` contains the following data:

```
{
    "isActive": true,
    "balance": "$2,186.98",
    "age": 33
}
```

Then, the `json` CLI can be used to turn the JSON data into a more compact format by using its option `--compact` while also ordering keys:

```
$ python -m json mydata.json --compact --sort-keys
```

The output produced is the following:

```
{"age":33,"balance":"$2,186.98","isActive":true}
```

The prefix `python -m json` is the command you use to run the CLI of the module `json`; `mydata.json` specifies the input file; `--compact` is a flag that suppresses all whitespace separation; and `--sort-keys` sorts keys in dictionaries alphabetically.

Run `python -m json --help` to see all the options and flags that the CLI supports.

156 – Custom t-string processing

t-strings, introduced in Python 3.14, offer a flexible and safe way of doing string formatting in contexts where care must be taken with the interpolated values.

For example, you can think about SQL injection attacks or having to escape some characters when interpolating strings that contain HTML.

When you create a t-string you must process it in order to create the final string object. For example, the variable `s` doesn't hold a string yet:

```
greeting = "Hey, there"
num = 3

s = t"He said {greeting} and waved {num} times."
#     ^^^^^^^^           ^^^^^^^^^^^     ^^^^^^^  # String literal parts.
#             ^^^^^^^^^^           ^^^^^          # Interpolated parts.
```

To process `s` into a string you can specify a function that iterates over the parts of the t-string and does whatever you need to each part. Parts that are interpolated values have an attribute `value` that is the original object being interpolated:

```
def auto_repr(tstring):
    parts = [
        part if isinstance(part, str) else repr(part.value)
        for part in tstring
    ]
    return "".join(parts)

print(auto_repr(s))
# "He said 'Hey, there' and waved 3 times."
#          ^^^^^^^^^^^^           ^          # Interpolated parts.
```

157 – `asyncio` introspection

You can use the `asyncio` CLI (new in Python 3.14) to inspect/debug running Python processes that use asynchronous tasks. For example, take this tiny example of an asynchronous program in the file `my_code.py`:

```
import asyncio

async def main():
    await asyncio.sleep(60)

asyncio.run(main())
```

If you run the program and the running process has the ID 12345, for example, then you can run

```
$ python -m asyncio pstree 12345
```

And you get a tree showing all currently-running `asyncio` tasks in a hierarchical format:

```
└── (T) Task-1
    └──  main /path/to/my_code.py:4
        └──  sleep /path/to/python/asyncio/tasks.py:702
```

The subcommand `ps` can be used instead of `pstree` to display this information in a table instead of in a tree.

158 – Grouping digits in the fractional part

You can *group digits in integers with underscores or commas*, but since Python 3.14 that you can also do the same to the digits of the fractional part of a number.

```
x = 12345.09876543

f"{x:.,}"  # '12345.098,765,43'
f"{x:._}"  # '12345.098_765_43'
```

The grouping can be specified independently for the digits before and after the decimal point by including a grouping character before or after the dot in the formatting specification:

```
f"{x:_.,}"  # '12_345.098,765,43'
```

Remember that if you only specify the grouping character, it will only group the digits of the integral part (see *"Formatting big numbers"*):

```
f"{x:_}"  # '12_345.09876543'
```

159 – Error handling with multiple types

When doing error handling with a `try: ... except: ...` block, you can handle different exception types in two different ways.

If you have two or more exception types that can be handled in the same way, you can group those exception types in the same `except` statement:

```
try:
    some_code()
except (ValueError, TypeError):
    do_something_universal()
```

In Python 3.14+, if you're not using the keyword `as`, you can omit the parentheses and write

```
# ...
except ValueError, TypeError:
    do_something_universal()
```

If different exception types need to be handled differently, you can have multiple `except` statements:

```
try:
    some_code()
except TypeError:
    do_something1()
except ValueError:
    do_something2()
```

160 – Parse dates from strings

You can use the class method `datetime.date.strptime` to parse a date from a string with the given format:

```
import datetime as dt

date_str = "26/10/2025"

when = dt.date.strptime(date_str, "%d/%m/%Y")
print(f"{when:%B %d, %Y}")  # October 26, 2025
```

This method is new in Python 3.14 for the class `datetime.date`, but the classes `datetime.datetime` and `datetime.time` already supported `strptime`.

161 – min-heaps and max-heaps

The module `heapq` has support for min-heaps and max-heaps, providing a number of functions to work efficiently with priority queues:

```
from heapq import (
    heapify,
    heappush,
    heappop,
    heappushpop,
    heapreplace,
)  # for min-heaps
```

For max-heaps, append the suffix `_max` to the name of each function:

```
from heapq import (
    heapify_max,
    heappush_max,
    heappop_max,
    heappushpop_max,
    heapreplace_max,
)  # for max-heaps
```

Support for min-heaps has been around for a while but support for max-heaps was only added in Python 3.14.

162 – Check for `None` functionally

Typically you use the expressions `... is None` and `... is not None` to check whether an object is or isn't `None`, respectively.

The module `operator` provides the functions `is_none` and `is_not_none` that perform the same checks:

```
from operator import is_none, is_not_none

print(is_none(None))  # True
print(is_not_none(None))  # False

print(is_none(73))  # False
print(is_not_none(73))  # True
```

These are useful when you're using higher-order functions.

163 – Remote interactive debugging

Since Python 3.14 that the module `pdb` has a CLI option that allows you to attach to a running Python process to debug interactively while it's running.

Suppose you have some buggy and complex business logic:

```
acc = 0

while True:
    acc += 1
```

You save this code in the file `myscript.py` and run it. The program never finishes, so you see it's running in the process with ID 12345 and you use the module `pdb` to attach to it:

```
$ python -m pbd -p 12345
```

Once you do, the debugger attaches to your program *while* it's running. Here, you can see you attached to the program as it was about to run the line `acc += 1`. Before you do, you may want to **p**rint the value of the variable `acc`:

```
> /path/to/script.py(4)<module>()
-> acc += 1
(Pdb) p acc
559642795
```

Now, you may want to go to the next line of the function and print the value of the variable `acc` again:

```
(Pdb) next
> /path/to/script.py(3)<module>()
-> while True:
(Pdb) p acc
559642796
```

164 – Chaining comparison operators

Python has plenty of comparison operators:

- `<`, `⩽`, `>`, `⩾`
- `==`, `!=`
- `in`, `not in`
- `is`, `is not`

All of these comparison operators can be chained together. For example, writing `floor ⩽ value ⩽ ceiling` is the same as writing `floor ⩽ value and value ⩽ ceiling`.

Chaining comparison operators can improve code readability, but only if

- used in sequences of `<` / `⩽` signs; or
- used in sequences of `>` / `⩾` signs.

Mixing different comparison operators will lead to code that is hard to parse. For example, `value in mylist == True` looks like `(value in mylist) == True`, but it's actually `value in mylist and mylist == True`, which will always be false if `mylist` is a list.

Further reading:

- Chaining comparison operators

165 – Controlled string splitting

The string method `str.split` has a parameter `maxsplit` that determines the maximum number of splits that will be performed:

```
>>> "a/b/c/d".split("/", 1)
['a', 'b/c/d']

>>> "a/b/c/d".split("/", 2)
['a', 'b', 'c/d']

>>> "a/b/c/d".split("/", 3)
['a', 'b', 'c', 'd']

>>> "a/b/c/d".split("/", 4)
['a', 'b', 'c', 'd']
```

The call to the method returns a list with the splits and possibly the remainder of the string, which means that the maximum length of the list returned is `maxsplit + 1`.

166 – `strftime` vs `strptime`

The methods `strftime` and `strptime` can be used to convert dates/times into strings and vice-versa. Here's a mnemonic to help you remember which is which:

- `strptime` has a "P" for "Parse date/time", so it'll accept a string and parse a date/time from it:

```
import datetime as dt

date = dt.date.strptime("2025-11-04", "%Y-%m-%d")
print(date)  # datetime.date(2025, 11, 4)
```

- `strftime` has an "F" for "Format date/time", so it'll accept a date/time and it will format it as a string:

```
print(date).strftime("%Y-%m-%d")
# 2025-11-04
```

167 – Private members

Python has a very well-known naming convention: names that start with a leading underscore `_` are considered "private".

This means the outside world has no business using them. For example, attributes and methods starting with `_` in a class mean they're for that class only. When that happens, it's common to see other methods of the class use those private names, like is the case for `drive`/`_start_engine` in the class `Car`:

```
class Car:
    def __init__(self, ...):
        self._serial_number = ...

    def _start_engine(self): ...

    def drive(self):
        self._start_engine()
        ...
```

Sometimes, private attributes/methods are still used inside the same module:

```
def _validate_serial_number(serial_no):
    ...

def inspect_car(workshop, car):
    if not _validate_serial_number(car._serial_number):
        ...
```

But if you're importing someone else's code, you should be very careful about using objects whose name starts with an underscore `_`.

168 – Truthy and Falsy

All Python objects have a Truthy/Falsy value which dictates how that object will behaving in a Boolean context.

Most objects are Truthy, with a few exceptions. For the built-in types, you usually have a specific value that is Falsy:

```
# Falsy values:
0        # zero
0.0      # zero, as a float
""       # empty string
[]       # empty list
{}       # empty dictionari
set()    # empty set
tuple()  # empty tuple
None     # None
```

Note how the Falsy values are "empty" or stand for "nothing". All other values of the same corresponding types are Truthy:

```
1, 2, -43              # any non-zero integer
0.1, 3.4, -3.1         # any non-zero float
"hi", "bye", "  "      # any non-empty string
[0], [[]], [1, 2]      # any non-empty list
{"a": "b"}             # any non-empty dictionary
{1, 2}, {False, True}  # any non-empty set
(0,), ([], True)       # any non-empty tuple
```

In case you are not sure, you can use the built-in `bool` to check the Truthy/Falsy value of an object:

```
print(bool(None))  # False
```

But when used in Boolean contexts, like `if` statements or in expressions with the keywords `not`, `and`, and `or`, you don't need the built-in `bool` because the Truthy/Falsy values of objects are used automatically.

Further reading:

- Truthy, Falsy, and `bool`

169 – Oxford comma

The function below turns a list of strings into a Human-readable enumeration that uses the Oxford comma:

```
def oxford_comma(strings):
    prefix = ", ".join(strings[:-1])
    ox_comma = "," if len(strings) > 2 else ""
    and_ = " and " if len(strings) > 1 else ""
    last = strings[-1] if strings else ""
    return prefix + ox_comma + and_ + last
```

Example usages:

```
print(oxford_comma(["like"]))
# like
print(oxford_comma(["like", "comment"]))
# like and comment
print(oxford_comma(["like", "comment", "subscribe"]))
# like, comment, and subscribe
```

The function `oxford_comma` demonstrates a useful technique for when you're building complex strings.

Instead of using multiple `if` statements to conditionally append parts of the string to the result, build the final string out of smaller fragments. Then, a fragment that is unnecessary can be set to the empty string.

This keeps your code flatter and cleaner.

170 – `assert` statements with custom message

The keyword `assert` takes an expression on its right and then raises an `AssertionError` if that expression doesn't evaluate to `True` (or Truthy):

```
x = 73
y = 42

assert y > x
```

```
...
AssertionError
```

The error message can be customised if you add a string in front of the expression being checked, separating the two with a comma:

```
x = 73
y = 42

assert y > x, f"{y=} isn't > {x=}..."
```

```
...
AssertionError: y=42 isn't > x=73...
```

Clear error messages make life easier for the person who's dealing with the error (which is likely to be "you" from the future), so take a second to write a useful error message that you'll be thankful for.

Just remember to never use `assert` for security-related or sensitive checks, since there are ways to "turn off" the assertions and have Python run without verifying them.

171 – Creating temporary files

You can use `tempfile.TemporaryFile` as a context manager that opens a temporary file that you can write to and read from:

```
import tempfile

with tempfile.TemporaryFile() as f:
    f.write(b"Hello ")
    f.write(b"world!")

    f.seek(0)
    print(f.read().decode())  # Hello world!
```

The default mode for the file is `"w+b"`:

- `"w+"` opens the file for writing *and* reading, so you can use `seek` to go back to the beginning of the file and read what you wrote to the file; and
- `"b"` opens the file in binary mode, meaning you'll write and read bytes.

Once the context manager is exited, the file is automatically deleted.

This is useful when testing functions that require file-like objects or as a buffer for when you're processing an amount of data so large that it doesn't fit into memory.

172 – Title case

Use the string method `title` to change a string's casing to title case:

```
print("star wars: the empire strikes back".title())
# Star Wars: The Empire Strikes Back
```

Title case will modify the case of each word so that its first letter is uppercase and all other letters are lowercase. For some reason, I always think that `title` will only make sure the first letter of each word is uppercase, but it will actively lowercase letters that are not at the start of a word:

```
print("CrAzY CAsInG".title())
# Crazy Casing
```

A "word" is a sequence of alphabetical characters, so you can have numbers of punctuation between the letters that `title` will still uppercase those as separate words:

```
print("1word2words3words".title())
# 1Word2Words3Words
```

173 – Using a list as a stack

When working with stacks, use the built-in `list` as the base class:

```
class stack(list):
    def put(self, value):
        self.append(value)

    def peek(self):
        return self[-1]
```

This allows you to inherit useful behaviours for free, like length:

```
s = stack()
s.put(1)
s.put(2)
s.put(3)

print(f"The stack has {len(s)} items.")
# The stack has 3 items.
```

Containment checks:

```
print(3 in s)  # True
popped = s.pop()
print(3 in s)  # False
```

Truthy/Falsy behaviour:

```
if s:
    print(f"{popped} was at the top.")
    print(f"Now it's {s.peek()}.")
# 3 was at the top.
# Now it's 2.
```

And more.

174 – Moving average

To compute the moving average of an iterable of values, use `collections.deque` to manage the window. The parameter `maxlen` that the `deque` has makes it very easy to manage the window and its elements:

```
from collections import deque
from itertools import islice

def moving_average(values, n):
    source = iter(values)
    window = deque(islice(source, n - 1), maxlen=n)
    averages = []
    for value in source:
        window.append(value)
        averages.append(sum(window) / n)
    return averages

moving_average([10, 20, 30], 2)
# [15.0, 25.0]
```

When you append to the window inside the `for` loop, the window will automatically evict its leftmost element to keep its size at `n` elements.

Further reading:

- Python `deque` tutorial

175 – Split from both sides

The string method `split` has a counterpart `rsplit` that starts splitting from the end of the string. If you split without restrictions, they behave the same way:

```
>>> "This is bananas".split()
['This', 'is', 'bananas']

>>> "This is bananas".rsplit()
['This', 'is', 'bananas']
```

However, if you specify a value for the parameter `maxsplit`, then `split` will find the first splits:

```
s = "This is bananas"
first, rest = s.split(maxsplit=1)
print(first)  # This
print(rest)  # is bananas
```

While `rsplit` will find the last splits:

```
s = "This is bananas"
rest, last = s.rsplit(maxsplit=1)
print(last)  # bananas
print(rest)  # This is
```

The string method `partition` also has a counterpart `rpartition` that partitions from the right.

These two methods that start operating from the end of the string are useful when you only want the last segment(s) of a string.

176 – Express permutations recursively

Recursion is disregarded by many as useless or "too fancy and theoretical to be practical", but that's very far from the truth. Recursion is a very natural way to express some algorithms and relationships and understanding it has many benefits.

`itertools.permutations` is an example of an algorithm that can be naturally expressed with recursion. Consider the permutations of the list `[0, 1, 2]`:

```
>>> from itertools import permutations
>>> list(permutations([0, 1, 2]))
[
    (0, 1, 2),
    (0, 2, 1),
    (1, 0, 2),
    (1, 2, 0),
    (2, 0, 1),
    (2, 1, 0),
]
```

The first two results are `(0, 1, 2)` and `(0, 2, 1)`, which are triples built by taking `0` and putting it in front of the permutations of the list `[1, 2]`:

```
>>> list(permutations([1, 2]))
[
    (1, 2),
    (2, 1),
]
>>> [(0,) + perm for perm in permutations([1, 2])]
[
    (0, 1, 2),
    (0, 2, 1),
]
```

This shows the recursive relation:

> "The **permutations** of a list of values start with the first value appended to the **permutations** of the *rest* of the values.

To build *all* the permutations, instead of just the first ones, we go through the list of all original values and, for each value, we prepend it to the permutations of all the other values. You can express this in code as such:

```
def perm_(values):
    # If there are no values, then there are no permutations; stop.
    if not values:
        yield ()
        return

    for idx, value in enumerate(values):
        rest = values[:idx] + values[idx + 1:]
        for sub_perm in perm_(rest):
            yield (value,) + sub_perm
```

177 – nwise

The iterable `itertools.pairwise` can be generalised to create an iterable that produces overlapping tuples of arbitrary length, instead of just pairs.

One of the best ways to generalise `pairwise` is by using `collections.deque` to manage the *window* of values that is going to be produced next:

```
from collections import deque
from itertools import islice

def nwise(values, n):
    iterator = iter(values)
    window = deque(islice(iterator, n - 1), maxlen=n)
    for value in iterator:
        window.append(value)
        yield tuple(window)
```

The line `window = deque(islice( ... ), ...)` seeds the `deque` with its first values. Then, the loop adds a value to the window and yields it as a tuple. `deque` is convenient to use here because of its parameter `maxlen`, that automatically makes sure the `deque` doesn't grow too large.

178 – Remove indentation from multiline strings

When you define a multiline string, you naturally get the indentation of the code that surrounds your string:

```
if show_instructions:
    instructions = """
        Instructions:
        Write Python.
    """
    print("Hey!")
    print(instructions)
```

```
Hey!

        Instructions:
        Write Python.
```

The instructions are indented because of the leading whitespace associated with the indentation of the `if` block they were defined in.

One way to fix this issue would be by deleting the indentation from the string yourself:

```
if show_instructions:
    instructions = """
Instructions:
Write Python.
"""
    print("Hey!")
    print(instructions)
```

```
Hey!

Instructions:
Write Python.
```

The problem with this solution is that the dedentend string breaks the flow of the code and makes it harder to understand the hierarchy of the code blocks.

A better fix for the indentation issue uses `textwrap.dedent`:

```
from textwrap import dedent

if show_instructions:
    instructions = dedent(
        """
        Instructions:
        Write Python.
        """
    )
    print("Hey!")
    print(instructions)
```

```
Hey!

Instructions:
Write Python.
```

`dedent` figures out how much indentation leading indentation is common to every line and removes it, allowing you to define multiline strings more conveniently.

179 – Immutable dataclasses

By default, dataclasses are mutable:

```
from dataclasses import dataclass

@dataclass
class Currency:
    name: str
    symbol: str

euro = Currency("euro", "€")
euro.name = "dollar"
print(euro)
# Currency(name='dollar', symbol='€')
```

The snippet above shows that the name of the currency `euro` was changed to `dollar`.

However, the decorator `dataclass` provides a convenient way to create immutable classes. To do that, just set `frozen=True` in the decorator:

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Currency:
    name: str
    symbol: str

euro = Currency("euro", "€")
euro.name = "dollar"
# dataclasses.FrozenInstanceError
```

If you try to mutate an immutable dataclass you get the error `dataclasses.FrozenInstanceError`.

180 – Fast exponentiation

Exponentiation, represented by `base ** exp` in Python, is shorthand for `exp - 1` multiplications of base with itself, but you can implement that computation without doing as many multiplications.

By using the binary expansion of the exponent, you can compute `base ** exp` with just $\log(exp)$ multiplications:

```
def fast_pow(base, exp):
    acc = 1
    while exp:
        exp, mod = divmod(exp, 2)
        if mod:
            acc *= base
        base **= 2
    return acc
```

This algorithm uses a technique called repeated squaring. The code block below shows how the values of `acc`, `exp`, and `base`, evolve during the computation of `fast_pow(3, 9)`:

```
acc   1 -> 3 ->  3  ->    3  -> 19683 -> done
exp   9 -> 4 ->  2  ->    1  ->     0
base  3 -> 9 -> 81  -> 6561  ->   ...
```

181 – Priority queues

The module `heapq` provides an interface to work with priority queues, where elements added to the queue have an associated priority and they are popped according to that priority.

Suppose you have a data class `Order` that represents orders at a pizza restaurant:

```
from dataclasses import dataclass

@dataclass(order=True)
class Order:
    priority: int
    order: str
```

The integer `priority` represents the order priority and the string `order` represents the food items ordered. Instances of the data class `Order` are orderable by the attribute `priority`, with `order` being used as a tie-breaker, thanks to the parameter `order=True` in the call to `dataclass`.

By using a priority queue, you can keep track of all orders and have them sorted by priority automatically:

```
import heapq

orders = []
heapq.heappush(orders, Order(5, "pizza"))
heapq.heappush(orders, Order(1, "2 large pizzas"))
heapq.heappush(orders, Order(10, "dessert"))

while orders:
    print(heapq.heappop(orders))
```

```
Order(priority=1, order='2 large pizzas')
Order(priority=5, order='pizza')
Order(priority=10, order='dessert')
```

Despite having been added *after*, the order for two large pizzas was popped *before* the order for a single pizza.

182 – Field default factory

Default mutable values create all kinds of problems, so using a mutable value as a default for a data class attribute raises an error:

```
from dataclasses import dataclass

@dataclass
class Exercise:
    name: str
    reps: int
    notes: list[str] = []  # Wrong!
# ValueError
```

The correct way to set the attribute `notes` to have an empty list by default is by using `dataclasses.field`:

```
from dataclasses import dataclass, field

@dataclass
class Exercise:
    name: str
    reps: int
    notes: list[str] = field(  # Right!
        default_factory=list
    )
```

The parameter `default_factory` can be set to a callable that should take no arguments. The callable is called *each time* the default value is needed.

183 – Deprecation warnings

Use the decorator `warnings.deprecated` to issue a deprecation warning when certain functions, methods, or classes are used:

```
from warnings import deprecated

@deprecated("Use bar instead.")
def foo():
    pass

def bar():
    pass

foo()
# DeprecationWarning: Use bar instead.
```

The code above shows that calling `foo` will issue a deprecation warning, since the function `foo` was decorated with the decorator `deprecated`.

Similarly, you can apply the decorator `deprecated` to a class:

```
from warnings import deprecated

@deprecated("Use D instead.")
class C:
    pass

class D:
    pass

c = C()
# DeprecationWarning: Use D instead.
```

184 – `accumulate` with a custom function

You can use a custom binary function with `itertools.accumulate`, allowing you to go through an iterable and computing successive accumulated values:

```
from itertools import accumulate
from operator import mul

print(list(accumulate(
    range(1, 6),
    mul,
)))
# [1, 2, 6, 24, 120]
```

You can also use a `lambda` function as the function argument to `accumulate`. In that case, the left argument is the accumulated value and the right argument is the next value from the iterable:

```
from itertools import accumulate
from operator import mul

print(list(accumulate(
    range(1, 6),
    lambda acc, v: 2 * acc + v
)))
# [1, 4, 11, 26, 57]
```

185 – Slicing mnemonic

If you think about sequence indices as the matching element separators, you can use that visual cue to understand how slicing works and what elements are included in a slice:

```
|   |   |   |   |   |   |   |
| S | L | I | C | I | N | G |
|   |   |   |   |   |   |   |
0   1   2   3   4   5   6   7
```

Using this visual cue, a slice includes all the elements between the corresponding separators:

```
print(s[3:6])  # 'CIN'
```

186 – Assigning to list slices

List slices can be assigned to:

```
my_list = [0, 1, 2, 3, 4, 5]
# my_list[2:4] --^^^^

my_list[2:4] = [98, 99]
print(my_list)  # # [0, 1, 98, 99, 4, 5]
```

A regular assignment to a list slice replaces the values that were in the slice.

The assigned value must be an iterable, but that's the only restriction. It doesn't even have to be of the same length as the slice on the left of the assignment operator:

```
my_list[2:4] = "abcd"
print(my_list)
# [0, 1, 'a', 'b', 'c', 'd', 4, 5]
```

The assignment above replaced the integers `98` and `99` with the four letters of the string `"abcd"`.

A consequence of the statement above, about the length of the iterables, is that using an empty iterable will delete the slice:

```
my_list[2:6] = []
print(my_list)  # [0, 1, 4, 5]
```

The slice `my_list[2:6]` was the list `['a', 'b', 'c', 'd']`, so assigning it to an empty iterable delete those values from the list `my_list`.

187 – Anatomy of a list comprehension

A list comprehension is made up of three sections:

1. **Data transformation**: The code that is transforming elements by applying a function or another expression to your values.
2. **Data source**: If you're transforming data, the data must come from somewhere.
3. **Data filter**: Optionally, you can add a data filter so you're only transforming *some* of the elements coming from the data source.

These three parts make up a general list comprehension. In code, it looks like this:

```
my_list = [
    data_transformation(value)  # 1
    for value in data_source    # 2
    if predicate(value)         # 3
]
```

Each line of the list comprehension above maps to one of the sections of a list comprehension. Here is a concrete example:

```
my_list = [
    n ** 2                          # 1
    for n in range(1000)            # 2
    if (n % 3 == 0) or (n % 5 == 0)  # 3
]
```

The list comprehension above

1. computes squares
2. of the integers from `0` to `999`
3. but only if the integer is divisible by `3` or `5`.

Further reading:

- List comprehensions 101
- List comprehensions cheatsheet

188 – Read file lines until a separator

To read the lines of a file until you hit a predetermined separator, you can use the special form of the built-in function `iter`, the method `readline`, and the string literal that matches the separator.

For example, suppose you have this file called `names.txt`:

```
Harry
Hermione
Ron
---
Potter
Granger
Weasley
```

If you want to read lines until you hit the separator `"---"`, you can use the building blocks mentioned:

```
first_names = []
with open("names.txt", "r") as f:
    for name in iter(f.readline, "---\n"):
        first_names.append(name.strip())

print(first_names[-1])  # Ron
```

The expression `iter(f.readline, "---\n")` creates an iterable that calls `f.readline` — to read a line from the file — repeatedly until `f.readline()` returns `"---\n"`, the line with the separator. Note how the separator ends with the newline character `"\n"`, since the method `f.readline` returns lines with the newline characters.

189 – Create pairs

To create pairs with elements from an iterable, you can use `itertools.product` for all ordered pairs:

```
from itertools import product

names = ["Harry", "Hermione", "Ron"]

for a, b in product(names, repeat=2):
    print(a, b)
```

```
Harry Harry
Harry Hermione
Harry Ron
Hermione Harry
Hermione Hermione
Hermione Ron
Ron Harry
Ron Hermione
Ron Ron
```

The pairs created included repeated elements, which you might want to filter by hand. Alternatively, you can use `itertools.combinations` for all unique pairs:

```
from itertools import combinations

for a, b in combinations(names, 2):
    print(a, b)
```

```
Harry Hermione
Harry Ron
Hermione Ron
```

Further reading:

- Module `itertools` overview

190 – Use set operations with dictionary keys

Dictionary keys also support set operations like union, intersection, and difference:

```
invited = {"Anne", "Ben", "Charles"}

arrivals = {
    "Anne": "7:59",
    "Ben": "8:03",
    "Dani": "8:15",
}

absent = invited - arrivals.keys()
print(absent)  # {'Charles'}

party_crashers = arrivals.keys() - invited
print(party_crashers)  # {'Dani'}
```

The snippet above shows that using a set operation with a dictionary keys object and a set is still a set.

191 – Frozen dataclasses

Frozen dataclasses are created by setting `frozen=True` in the decorator `dataclass`:

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Point:
    x: int
    y: int
```

By being set to frozen, data classes become immutable and hashable, making them usable as dictionary keys or as set elements:

```
p = Point(0, 0)

my_dict = {p: "origin"}
print(my_dict)
# {Point(x=0, y=0): 'origin'}
```

Surprisingly, frozen dataclasses pay a tiny performance penalty.

192 – Break from nested loops

Breaking out of nested `for` loops requires auxiliary variables and conditional statements:

```
done = False
for n in range(1, len(switches) + 1):
    for group in combinations(switches, n):
        state = simplify(group)
        if state == target:
            print("found!")
            done = True
            break
    if done:
        break
```

You need to use the Boolean flag `done` and an extra `if done` statement to be able to break out of the two loops once you find the value of `state` for which `state == target`.

To improve this code, you can refactor the looping logic into a generator:

```
def groups_of_switches(switches):
    for n in range(1, len(switches) + 1):
        for group in combinations(switches, n):
            yield group
```

The generator should contain only the loops and no other logic. By extracting the loops into a generator, you can refactor the original code so it's flatter:

```
for group in groups_of_switches(switches):
    state = simplify(group)
    if state == target:
        print("found!")
        break
```

The flatter code is easier to reason about and it's easier to break out of this loop because it's no longer nested.

193 – Typing generators

Generator type hints are cumbersome to write because the generic type `Generator` takes three arguments:

```
from collections.abc import Generator

def simple_generator(stop: int) -> Generator[int, None, None]:
    for x in range(stop):
        yield x ** 2
```

The three types indicate the type of the yielded values, the type of the send values, and the type of the return.

In Python 3.13, the send and return types were changed so that they default to `None`, allowing you to simplify many usages of `Generator`:

```
from collections.abc import Generator

def simple_generator(stop: int) -> Generator[int]:
    for x in range(stop):
        yield x ** 2
```

The omission of the second and third types indicate that they are `None`.

194 – Type-safe dictionaries

You can add type safety to dictionaries with a fixed structure by using `typing.TypedDict`, which you define by specifying the keys and the types of the values:

```
from typing import TypedDict

class Config(TypedDict):
    debug_mode: bool
```

The typed dictionary `Config` is a dictionary that has the key `"debug_mode"`, and that key maps to a Boolean value. Then, if you use the type `Config` in your code, your static type checker will ensure you're using a dictionary that contains the required keys:

```
def run(..., c: Config) -> None:
    if c["debug_mode"]:
        print("run starting")
    ...
```

The key access `c["debug_mode"]` type checks because the dictionary `c` is a `Config` dictionary, which has the key `"debug_mode"` that maps to a Boolean value.

If you didn't use a typed dictionary, you would probably say that `c` is a dictionary that maps strings to Booleans:

```
def run(..., c: dict[str, bool]) -> None:
    if c["debug_mode"]:
        print("run starting")
    ...
```

In this case, there is an *infinite* number of dictionaries that would pass type checking if passed to the function `run` and that would raise a `KeyError` because they don't have the correct key.

195 – ID generator

Use `itertools.count` to generate unique sequential IDs:

```
from itertools import count

class Tab:
    _id_generator = count()

    def __init__(self):
        self.tab_id = next(self._id_generator)
```

By keeping a reference to `_id_generator` and reusing it inside `Tab.__init__`, every time you create a tab you get a new ID:

```
tab1 = Tab()
tab2 = Tab()
tab3 = Tab()

print(tab1.tab_id)  # 0
print(tab2.tab_id)  # 1
print(tab3.tab_id)  # 2
```

196 – Drop into a debugger

The built-in function `breakpoint` stops your program and drops you into a debugger at the call site. For example, suppose you write this program, that defines two variables and calls `breakpoint` *before* the final call to `print`:

```
x = 73
y = 42

breakpoint()

print("Program finished")
```

By running this program, you'll be dropped into a debugger between the variable assignments and the call to `print`. By default, `breakpoint` opens pdb:

```
-> breakpoint()
(Pdb) p x
73
(Pdb) p y
42
(Pdb) c
Program finished
```

When you're done debugging, you can type `c` (short for **c**ontinue) to exit the debugger and run the remainder of the program.

197 – Fully consume an iterator

To fully consume an iterator without wasting any memory, use an instance of `collections.deque` with its maximum length set to `0`:

```
from collections import deque

squares = (n ** 2 for n in range(100))

deque(squares, maxlen=0)

for sq in squares:
    print(sq)
# No output!
```

Running the code above produces no output because the generator `squares` is fully consumed by the call to `deque`.

The `deque` approach is faster than an empty Python loop because `deque` is implemented in `C` whereas the loop runs at the Python level.

198 – Data class with default mutable value

If you try to set a data class field to a mutable value, you get an error:

```
from dataclasses import dataclass

@dataclass
class Error:
    lineno: int
    error_type: str
    notes: list[str] = []
```

```
ValueError: mutable default <class 'list'> for field notes is not allowed: use␣
↪default_factory
```

The decorator `dataclass` validates each field in your data class and it preemptively throws an error when it sees the list `[]` being used as the default value for the field `notes`. The reason you get the error is because it is *highly* likely that your code will have bugs if the error wasn't raised.

The correct way to have a list be the default value is by using `dataclasses.field` and the parameter `default_factory`:

```
from dataclasses import dataclass, field

@dataclass
class Error:
    lineno: int
    error_type: str
    notes: list[str] = field(
        default_factory=list
    )
```

The parameter `default_factory` accepts a zero-argument function that is called whenever you need to populate your field with a default.

199 – typing.NewType

Sometimes you are dealing with lots of data of the same type (e.g., strings) but they all have different meanings. For example, a person's name is very different from their email. You can use `NewType` to create types that differentiate the different meanings so that the type checker can help you find bugs in your program.

Can you spot the bug in the code below?

```
def send_email(email: str, body: str) -> bool:
    msg = EmailMessage(to=email, body=body)
    return msg.send()

data: tuple[str, str, str] = (
    "John",
    "john@example.com",
    "Hi John!",
)

email, name, body = data
send_email(email, body)  # Type checks!
```

The problem is that the unpacking `email, name, body = data` is wrong and you're thus passing the person's name as the email address to the function `send_email`. The type checker could find this bug for you if you had a `NewType` specifically for the email field:

```
from typing import NewType

Email = NewType("Email", str)
```

Now you update the signature of `send_email` to use `Email`:

```
def send_email(email: Email, body: str) -> bool:
    ...
```

Finally, you update the type hint of `data` to signal where the `Email` goes:

```
data: tuple[str, Email, str] = (
    "John",
    Email("john@example.com"),
    "Hi John!",
)
```

You use `Email` around the string so that the type checker sees that string as an email. Now, the static type checker would catch the mistake in the unpacking that follows.

Note that `NewType` does not do any runtime validation, so something like `Email(73)` is perfectly valid at runtime (but not at static type-checking time).

200 – One-shot data compression

The module `compression`, new in Python 3.14, provides a convenient way to do one-shot data compression. There are *5 different algorithms to choose from*, but you can default to the Zstd if you don't have any particular reason to pick one over another.

To compress data, make sure you're working with bytes and call the function `compress`:

```
from compression import zstd

data = ("Hello, world!" * 1000).encode()

compressed = zstd.compress(data)
```

You can see this compresses the data to only occupy 0.2% of its original size:

```
print(len(compressed))  # 31
print(len(compressed) / len(data))  # 0.0023846153846153848
```

You can decompress the data with the aptly-named function `decompress`:

```
print(zstd.decompress(compressed) == data) # True
```

Further reading:

- Module `compression` overview

201 – Check if a number is a power of 2

You can use a clever bitwise AND (`&`) to quickly and efficiently check if a number is a power of 2:

```
def is_power_of_two(n):
    return n > 0 and (n & (n - 1)) == 0
```

This is much more efficient than using something like `math.log2`.

The diagram below helps visualise this operation for `n = 64`:

```
    n = 64 = 1000000
n - 1 = 63 = 0111111
   64 & 63 = 0000000
```

You can also quickly test the function `is_power_of_two` on every single integer from 0 up to whatever limit you establish:

```
powers_of_two = {2 ** n for n in range(20)}

assert set(filter(is_power_of_two, range(2 * max(powers_of_two)))) == powers_of_two
```

202 – Base64 encoding

Base64 encodes binary data in a way that makes it safe to send over email, use on URL parameters, as HTTP POST requests, and more. Python provides the module `base64` to work with it:

```
from base64 import b64encode, b64decode

data = "What's up?"
encoded = b64encode(data.encode())

print(encoded)  # b'V2hhdCdzIHVwPw=='

print(b64decode(encoded).decode())  # What's up?
```

It's important to note that the functions from the module `base64` expect and return bytes objects.

(Base64 derives its name from the 64 characters it uses to encode data.)

203 – Cached properties

You can use `functools.cached_property` to create a property that is computed once and then cached for the lifetime of the instance where it is defined. This is a good approach for attributes that are expensive to compute because it defers the cost of computing them to as late as possible. If the attribute is never needed, you never have to compute it.

Here's an example usage:

```
from functools import cached_property
from math import sqrt

class Int:
    def __init__(self, n):
        self.value = n

    @cached_property
    def is_prime(self):
        if not self.value % 2:
            return False
        div, stop = 3, sqrt(self.value)
        while div <= stop:
            if not self.value % div:
                return False
            div += 2
        return True
```

To see it in action, you can create an instance of `Int` with a large-enough prime number:

```
# 10000000000th prime:
i = Int(252097800623)

print(vars(i))  # {'value': 252097800623}

print(i.is_prime)  # True

print(vars(i))
# {'value': 252097800623,
# 'is_prime': True}

print(i.is_prime)  # True
```

The snippet above also shows that the cached property will set an attribute with the same name on the instance. While it's there, checking `i.is_prime` will access the attribute directly instead of running the property. If you delete the attribute with `del i.is_prime`, then you can force the cached property to run again.

204 – Reading and writing JSON

You can use the module `json` to read and write data in the JSON format, which is very suitable to represent the most common Python built-in types like lists and dictionaries, strings, integers and floats, and Boolean values.

To write data in the JSON format to a file, use the function `json.dump`:

```
import json

data = {
  "name": "Rodrigo",
  "newsletters": 2,
}

with open("data.json", "w") as f:
    json.dump(data, f)
```

The call to `json.dump` writes a file that looks like this:

```
{"name": "Rodrigo", "newsletters": 2}
```

You can retrieve the data and convert it back to the appopriate Python types with `json.load`:

```
import json

with open("data.json", "r") as f:
    data = json.load(f)

print(data)
```

```
data = {
  "name": "Rodrigo",
  "newsletters": 2,
}
```

205 – Parse, don't validate

Whenever you are using validator functions you run the risk of forgetting to call the validator. This means you'll be passing around data that might be invalid.

You can prevent this from happening by writing a "parser" function instead of a validator. This "parser" accepts the data and, if it's valid, returns it as a new type. Here's an example for email addresses:

```
from typing import NewType

Email = NewType("Email", str)

def parse_email(email: str) -> Email:
    # Check if it's valid …
    return Email(email)

def login(email: Email): ...
```

Now, if you try to call `login` without passing through `parse_email`, your static type checker will complain:

```
email = input("Enter an email >> ")
login(email)  # Type error!
```

The fix is calling `parse_email` first:

```
email = input("Enter an email >> ")
email = parse_email(email)
login(email)
```

This is some example text. This text is in **bold** and this is *italics*.

I have a link.

Call the built-in function `print` and do this:

```
import functools

print(functools)
```

206 – `typing.Self`

When a method returns the argument `self`, use `typing.Self` to annotate the return type of the method:

```
from typing import Self

class SomeClass:
    def some_method(self) -> Self:
        ...
        return self
```

This is especially useful if `AnotherClass` inherits from `SomeClass`:

```
class AnotherClass(SomeClass):
    pass
```

Because of `Self`, the return type of `AnotherClass.some_method` is inferred to be the subclass itself:

```
from typing import reveal_type

reveal_type(AnotherClass().some_method())  # AnotherClass
```

If you didn't use `Self`, the return type would be the parent class:

```
from typing import reveal_type

class SomeClass:
    def some_method(self) -> SomeClass:
        ...
        return self

class AnotherClass(SomeClass):
    pass

reveal_type(AnotherClass().some_method())  # SomeClass ?!
```

207 – Regex groups with default values

A regex match has the method `groups` that returns a tuple with all of the groups that the pattern contained:

```
import re

pattern = r"""(?x)
    (\d+)  # 1 or more digits
    \.?    # An optional .
    (\d+)? # More optional digits
"""

m = re.match(pattern, "24.5")

print(m.groups())
# ('24', '5')
```

However, if one or more of the groups did not participate in the match, it will be replaced with the default value `None`:

```
m = re.match(pattern, "24")

print(m.groups())
# ('24', None)
```

Alternatively, you can provide your own value for the groups that didn't participate:

```
print(m.groups("0"))
# ('24', '0')
```

208 – itertools.pairwise

`itertools.pairwise` accepts an iterable as its only argument and produces a sequence of overlapping pairs of consecutive elements:

```
from itertools import pairwise

queue = ["Harry", "Hermione", "Ron"]

for front, back in pairwise(queue):
    print(f"{front} is directly in front of {back}.")
# Harry is directly in front of Hermione.
# Hermione is directly in front of Ron.
```

209 – `itertools.tee` splits iterators

Iterators can only be traversed once, like is the case with generators:

```
squares = (x ** 2 for x in range(3))

for sq in squares:
    print(sq, end=", ")
# 0, 1, 4,

for sq in squares:
    print(sq, end=", ")
# <no output>
```

The second loop produces no output because the generator was exhausted during the first loop.

By using `itertools.tee`, you can work around this limitation, since `itertools.tee` allows you to create 2+ independent iterators out of a single iterator:

```
from itertools import tee

squares = (x ** 2 for x in range(3))
sq1, sq2 = tee(squares, 2)

for sq in sq1:
    print(sq, end=", ")
# 0, 1, 4,

for sq in sq2:
    print(sq, end=", ")
# 0, 1, 4,
```

This time, the second loop produces the same output as the first loop because both loops traverse two different iterators (`sq1` and `sq2`), that were returned by `itertools.tee`.

210 – Convert data to a JSON string

The function `dumps` from the module `json` can be used to convert data into a JSON string:

```
import json

data = {"key1": True, "key2": [73, 42, 10]}

dumped = json.dumps(data)
print(type(dumped), dumped)
# <class 'str'> {"key1": true, "key2": [73, 42, 10]}
```

The value `dumped` that is printed looks almost the same as the variable `data`, but you can see that the Boolean value `True` was converted to `true`, since `True` is not a valid JSON value.

This is useful if you need to pass JSON data to other functions or processes but if your goal is to write the JSON data to a file, use the function `dump` instead:

```
import json

data = {"key1": True, "key2": [73, 42, 10]}

with open("myfile.json", "w") as f:
    json.dump(data, f)
```

211 – `TypeAlias` vs `NewType`

While similar at first sight, `TypeAlias` and `NewType` are fundamentally different. `TypeAlias` creates a synonym to a complex type with the purpose of simplifying long or cumbersome type signatures:

```
type P = tuple[int, int]

def add_points(p1: P, p2: P) -> P:
    x1, y1 = p1
    x2, y2 = p2
    return (x1 + x2, y1 + y2)
```

By creating the type alias `P`, you don't have to write `tuple[int, int]` three times in the signature of the function `add_points`.

`P` being a type *alias*, `P` and `tuple[int, int]` are completely interchangeable:

```
p: tuple[int, int] = (1, 2)
reveal_type(add_points(p, p))
# Revealed type: tuple[int, int]
```

Despite being typed as `tuple[int, int]`, the variable `p` can be used as the argument for `add_points`, and despite being typed as returning a value of type `P`, the type of the return value of the function `add_points` is revealed to be `tuple[int, int]`.

On the other hand, `NewType` creates a new type that can be seen as a specialised version of the original type:

```
from typing import NewType

Length = NewType("Length", int)
Area = NewType("Area", int)

def square_area(l: Length) -> Area:
    return Area(l * l)
```

Despite being both integers, a length is semantically different from an area, and the new types `Length` and `Area` can encode that information.

By using new types, the type checker will warn you if you try to mix different types of integers up:

```
a = square_area(3)  # Type error.
a2 = square_area(a)  # Type error.
```

The call `square_area(3)` fails type checking because the function `square_area` expects explicitly a `Length`, and not just a normal integer. Similarly, `square_area(a)` fails type checking because `a` is of type `Area`, which is semantically different from the `Length` that the function expects.

212 – Patterns in `glob` search

The method `glob` of `pathlib.Path` can look for file names that match patterns using character ranges. The syntax for a character range is the same as in regular expressions, and it's the square brackets `[ ... ]` with two characters separated by a hyphen. For example, the character range `[0-9]` matches any of the ten digits in the string `"0123456789"`.

Suppose your current working directory has these files:

```
- 01_ideas.txt
- 02_todo.txt
- 03_cover.png
- draft.txt
```

The first three files have a numeric prefix but only two of them are text files (`.txt`). If you wanted to find all text files with a numeric prefix, you could use two character ranges to do that:

```
import pathlib

for txt_file in pathlib.Path().glob("[0-9][0-9]*.txt"):
    print(txt_file.name)
# 01_ideas.txt
# 02_todo.txt
```

The character ranges `[0-9][0-9]` match the initial numeric prefix and the asterisk `*` matches an arbitrary file name. The final `.txt` matches the literal characters `".txt"`.

Note that the patterns that `glob` supports may *look like those of regular expressions*, but they are not the same thing. `glob` supports a much more limited set of patterns.

213 – Expanding regex matches

Regex matches can be used to expand strings that reference matching groups, allowing you to quickly and conveniently format strings with data extracted from a regex.

You start by getting a regex match using any of the functions that the module `re` provides:

```
import re

m = re.search(r"(\d+)\.(\d{2})", "Total: 12.45€")
```

The regex above extracts the number `12.45` and uses two groups to target the integer and decimal parts of the number.

Then, the method `expand` of the match can be used to format any string that references the matching groups:

```
print(m.expand(r"The total is \1 euros and \2 cents."))
# The total is 12 euros and 45 cents.
```

The escape sequences `\1` and `\2` are actually references to the matching groups of the match `m`. To make it easier to include the references, a raw string `r" ... "` is used, otherwise the backslashes would have to be doubled.

214 – `reveal_type`

The module `typing` provides a debugging aid, the function `reveal_type`. When a static type checker encounters it, the static type checker emits a note with the *inferred* type of the expression passed into `reveal_type`:

```
def s() -> str:
    return 3

reveal_type(s())
# Revealed type is "builtins.str"
```

The function `reveal_type` doesn't even need to be imported and the static type checker infers that `s()` is a string because of the return type `-> str`. (The static type checker also complains about the fact that we're returning an integer from inside `s` while the function should return a string, but you'll see why I'm doing that in a second.)

At runtime, the function `reveal_type` prints the runtime type of the argument to `sys.stderr` and returns the argument unchanged:

```
from typing import reveal_type

def s() -> str:
    return 3

reveal_type(s())
# Runtime type is 'int'
```

At runtime, the type of `s()` is `int` because the function `s` returns the integer 3. For the code to work at runtime, `typing.reveal_type` *must* be imported since it's not a built-in.

215 – Implementing `pairwise` with `tee`

Before being added to Python 3.10, `itertools.pairwise` was commonly implemented in terms of `itertools.tee`:

```
def my_pairwise(it):
    it1, it2 = tee(it, 2)
    next(it2)
    yield from zip(it1, it2)
```

This short implementation shows how elegantly the different built-ins and the iterators from `itertools` can be combined to express new ideas.

216 – Setting the exit code

For scripts, command line interfaces, and other code that is typically run in the terminal, use `sys.exit` to set the exit code to an appropriate value.

An exit code of `0` means the program terminated successfully:

```
import sys

# ...

sys.exit(0)
```

Any other integer value (from `1` to `255`) will tell the terminal that the program ended with an error:

```
import sys

# ...

sys.exit(42)  # Error code 42.
```

It is up to your program to assign meaning to each error code, in case it's relevant.

A program that terminates with an exception automatically gets the error code `1`:

```
# myscript.py

1 / 0
```

The file `myscript.py` has the single line `1 / 0`, which raises an exception. Running it in the terminal, you see the exception:

```
$ python myscript.py
ZeroDivisionError: division by zero
$ echo $?
1
```

The command `echo $?` prints the exit code of the previous command, so the result of `1` shows that Python terminated with the exit code `1`.

217 – Parse JSON string into object

You can parse a JSON string into a native Python object by using the function `loads` from the module `json`:

```
import json

data = '{"hey": [42, null, false]}'

d = json.loads(data)
print(d)
```

```
{'hey': [42, None, False]}
```

The function `loads` stands for _load s_tring, and is useful when the JSON data doesn't come from a file, for example when it's a payload from an API call.

Themed index

#

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z

Conclusion

I hope you learned a thing or to by going through this book. If you have any feedback, email me at rodrigo@mathspp.com or find me on social media:

- LinkedIn: https://linkedin.com/in/rodrigo-girão-serrão
- BlueSky: https://bsky.app/profile/mathspp.com
- YouTube: https://www.youtube.com/@mathsppblog
- X/Twitter: https://x.com/mathsppblog