Python drops

Rodrigo Girão Serrão

25-05-2025

Contents

Introduction

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!

Tips

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("0xff", 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=" ")
# 3 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("AAABOAA")
))  # [('A', 3), ('B', 1), ('O', 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__`.

Themed index

A

B

C

D

E

F

G

I

K

L

M

N

O

P

R

S

T

U

V

Z

-

- `__file__`: 20
- `__init__`: 58
- `__init_subclass__`: 26
- `__new__`: 58

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