zip's keyword argument strict

Set `strict=True` if you expect the arguments to have the same length. `zip` will error if they do not.

```
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-2>", line 1, in <module>
    for first_name, last_name in zip(firsts, lasts, strict=True):
                                 ~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
ValueError: zip() argument 2 is shorter than argument 1
```

The error only happens when `zip` reaches the end of one, but not all, of the arguments.

Rodrigo 🐍🚀
🦋 @mathspp.com

Case-insensitive string comparisons

When comparing strings in a case-insensitive manner, use the method `casefold`, which is designed specifically for this task.
You need `casefold` because some characters across different languages are... kinda funky!

This string is lowercase...
But converting to upper and back to lowercase doesn't roundtrip!

```
print("straße".lower())
# straße

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

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

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

💡 The methods `lower/upper` can be used if you are ***sure*** you're only working with ASCII.

Rodrigo 🐍🚀
🦋 @mathspp.com

Type unions with pipe in `isinstance`

Types can be combined with the pipe operator `|` to create a type union.

This works, for example, inside `isinstance`, to make it more convenient to check if a value has one of 2+ types.

This checks if `x` is an `int` OR a `float` OR a `complex` number.

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

print(is_number(34))  # True
print(is_number("hey"))  # False
```

💡 Only available in Python 3.10+.

Rodrigo 🐍🚀
🦋 @mathspp.com

Parsing integers from different bases

The built-in `int` can parse integers from any base from binary to base 36 (including octal, decimal, and hexadecimal).

`0` tells int to interpret the string as a string literal (which uses the prefix to guess the base).

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

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

Rodrigo 🐍🚀
🦋 @mathspp.com

First element that satisfies a condition

Need the first element of an iterable that satisfies some condition?
Use a generator expression and the built-in `next`.

The built-in `next` accepts a fallback to return if it finds nothing. If you don't specify the fallback, you get a `StopIteration` exception instead.

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

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

Rodrigo 🐍🚀
@mathspp.com

Last element that satisfies a condition

Need the last element of an iterable that satisfies some condition?
Use a `deque` from the module `collections`.

```
from collections import deque

important_numbers = [42, 73, 10, 16, 0]
print(  # Last even number.
    deque(
        (n for n in important_numbers if n % 2 == 0),
        maxlen=1,
    )
).pop()  # 0

large = (n for n in important_numbers if n > 100)
if (d := deque(large, maxlen=1)):
    print(d.pop())
else:
    print("No large numbers present!")
```

If there is no element in the iterable that satisfies the condition, the `deque` will be empty and you can't pop from it (you get an `IndexError`).

Rodrigo 🐍🚀
@mathspp.com

Unique elements from a list

```python
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
```

You can get the unique elements of a list (or of another iterable) with the built-in set type. The result is in an arbitrary order.

If you need to preserve the original order, you can use `dict.fromkeys` instead.

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

💡 Both examples only work with hashable

Rodrigo 🐍🚀
🦋 @mathspp.com

How to schedule program cleanup

Use the function `register` from the module `atexit` to schedule a function to run at program exit.

Your cleanup function will likely clear resources you're using, like database connections.

```python
import atexit

def cleanup():
    """Clean up program resources."""
    db.close_connection()
    print("Cleaned up!")

atexit.register(cleanup)
```

Registered functions run even if the program terminates with an exception.

💡 `register` can be used as a decorator.

Rodrigo 🐍🚀
@mathspp.com

map with multiple arguments

The built-in `map` can accept 2+ iterable arguments if the function it's mapping takes 2+ arguments.

```
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 makes `map` a viable alternative to list comprehensions/generator expressions in some situations.

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

💡 In Python 3.14, `map` gets a keyword argument `strict`, like `zip`'s.

Rodrigo 🐍🚀
🦋 @mathspp.com

Remove punctuation from a string

The string method `translate` lets you replace (or delete) multiple characters of a string in one go. (Kind of like multiple calls to the method `replace`.)

```
import string

s = "Hello, world!"
print(s.translate(str.maketrans("", "", string.punctuation)))
# Hello world
```

The string method `translate` expects a "translation table", which we build with the auxiliary class method `maketrans`. The third argument is a string of characters we want to delete.

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

Rodrigo 🐍🚀
🦋 @mathspp.com

Count characters in a file

Use `chain`, from the module `itertools`, to iterate over the characters of a file. Used with `Counter`, from the module `collections`, lets you count the characters in a file.

```python
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)]
```

`f` lets you iterate over the lines of the file, so `chain.from_iterable(f)` lets you iterate over the characters in each line of the file in turn.

Rodrigo 🐍🚀
🦋 @mathspp.com

Run-length encoding

Use `groupby`, from the module `itertools`, to group consecutive equal elements into groups.
When used right, `groupby` has many useful use cases.

```python
from itertools import groupby

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


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

If the groups are very long, this way of computing the length of the group might be a waste of memory.

💡 `groupby` also accepts a keyword argument `key`, if you want to apply a transformation to the values before grouping.
E.g., using `key=str.casefold` would group equal characters regardless of their casing.

Rodrigo 🐍🚀
🦋 @mathspp.com

String prefixes and suffixes

Strings have a method `startswith` to check if a string has a certain prefix and a method `removeprefix` (3.9+) to remove the given prefix from the string.

```python
string = "Hello, world!"

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

```python
string = "Hello, world!"

print(string.startswith("Hey"))
# False
print(string.removeprefix("Hey"))
# 'Hello, world!'
```

If the prefix isn't there, `removeprefix` leaves the string unchanged.

💡 Use these methods instead of slicing for readability.

💡💡 Use `endswith` and `removesuffix` to work with suffixes.

Rodrigo 🐍🚀
@mathspp.com

`match` multiple options

In structural pattern matching (`match` statement, Python 3.10+) you can use the vertical bar to separate multiple options.

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

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

Rodrigo 🐍🚀
@mathspp.com

Round to “pretty” whole numbers

The built-in `round` accepts negative integers as its argument. A negative integer rounds to a power of 10, which tends to be a “nice”/“pretty” whole number.

```
price = 1374
hundreds = round(price, -2)

print(
    f"It cost roughly {hundreds} dollars."
)
# It cost roughly 1400 dollars.
```

The integer without the sign gives the number of 0s.

Rodrigo 🐍🚀
@mathspp.com

type statements

Python 3.12 introduced type statements: a convenient way of creating type aliases that can also be made generic.

```python
from typing import TypeAlias, TypeVar

T = TypeVar("T")

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

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

The type variable for the generic doesn't need to be defined separately.

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

✅ Both versions type-check correctly.

Rodrigo 🐍🚀
@mathspp.com

Create context managers with `contextlib.contextmanager`

Use `contextlib.contextmanager` as a decorator to turn a generator into a context manager.

```python
from contextlib import contextmanager

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

What you yield (if anything) can be captured by the "`as ...`"

```python
with my_open(".zshrc", "r") as f:
    print(len(list(f)))  # 129
```

Regardless of whether there's an error or not, the file will be closed.

Rodrigo 🐍🚀
mathspp.com/blog

Immutable dictionary

You can use `MappingProxyType` to create an immutable / read-only dictionary.

```python
from types import MappingProxyType

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

Access values with keys, as usual.

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

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

Modifying values or creating new key/value pairs raises a `TypeError`.

💡 Don't keep references to the underlying dictionary, because that's still mutable...

Rodrigo 🐍🚀
mathspp.com/blog

self-debugging f-strings

Include an equals sign = at the end of a formatted value and the f-string will print the expression you are formatting (in code) together with its value.

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

# Method title: name.title() = 'Rodrigo'
```

Code

Value

💡 The spaces around the equal sign aren't necessary but the result looks better with those.

Rodrigo 🐍🚀
mathspp.com/blog

Dunder attribute `__file__`

In a Python file, the dunder attribute `__file__` gives you the full path of your file.

tool.py

```python
print(__file__)
# /Users/rodrigogs/Documents/tmp/tool.py
```

Useful, for example, to locate resources that are "next" to your script.

```python
from pathlib import Path

RESOURCES = (Path(__file__).parent / "res").resolve()
```

💡 __file__ isn't *always* available. For example, in Jupyter notebooks.

Rodrigo 🐍🚀
mathspp.com/blog

Current date and time

The module `datetime` has two functions to produce a `date` object with the current date and a `datetime` object with the current date and time.

```python
import datetime as dt

today = dt.date.today()
print(f"{today:%c}")
# Tue Apr  1 00:00:00 2025

now = dt.datetime.now()
print(f"{now:%c}")
# Tue Apr  1 13:43:44 2025
```

The format specifier %c displays the date (and time) in a locale-appropriate way.

Rodrigo 🐍🚀
mathspp.com/blog

Set operations with `dict.keys()`

Dictionary `keys` objects support set operations.

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

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

```python
# 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', 'blue', 'yellow'}
```

Some operations supported include difference, union, and intersection.

Rodrigo 🐍🚀
mathspp.com/blog

Chain multiple dictionaries

You can use `ChainMap` from the module `collections` to create a unified view over a hierarchy of dictionaries.

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

```
local = {

    "theme": "dark",

}
```

```
user = {
    "user": "rodrigo",


}
```

```
from collections import ChainMap

settings = ChainMap(user, local, default)

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

In this example, we can read settings from the user, from the local installation, or the defaults. Each level can be edited individually.

Rodrigo 🐍🚀
mathspp.com/blog

Longest word in a string

You can find the longest word in a string by using the built-in `max` and by specifying the built-in `len` as the key.

```
s = "These are just some sensational words"

print(max(s.split(), key=len))
# sensational
```

Using the built-in `min` would produce the shortest word instead (are) and using the built-in `sorted` would order the words by length.

Rodrigo 🐍🚀
mathspp.com/blog

Dynamic attribute manipulation

The built-ins `getattr`, `setattr`, and `delattr`, can be used to manipulate attributes dynamically.
You'll rarely use these, but when you need them, they are very powerful.

```python
class Colour:
    pass

c = Colour()

setattr(c, "r", 255)
setattr(c, "g", 125)
setattr(c, "b", 0)

print(c.g)  # 125
```

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

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

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

Default value.

The first two arguments are the object and the name of its attribute being manipulated.

⚠️ This shows how the functions work but not necessarily how you'd use them in “real life”.

Rodrigo 🐍🚀
mathspp.com/drops

Notify parent class when subclassing

The class method `__init_subclass__` can be used to notify a class when it's subclassed. Effective for some metaprogramming without metaclasses.

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

The new subclass is the `cls` passed in.

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

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

The keyword arguments given to the new class are passed to `__init_subclass__` as well.

Rodrigo 🐍🚀
mathspp.com/drops

Enforce keyword args for options

Use a single asterisk * between commas to force all following arguments to be keyword only.
Particularly helpful for options or “config-like” arguments.

```python
def get_temperature(room, *, unit="celsius"):
    pass

get_temperature("bedroom", "fahrenheit")  # ⚠️ TypeError

get_temperature("bedroom", unit="fahrenheit")  # ✅
```

The argument `unit` must be passed as a keyword argument.

Rodrigo 🐍🚀
mathspp.com/drops

Flag enumerations

The module `enum` contains a `Flag` type 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 = Color.RED | Color.BLUE

print(Color.GREEN in purple)  # False
print(list(purple))
# [<Color.RED: 1>, <Color.BLUE: 4>]
```

Purple is red with blue.

Is the flag GREEN set?

What flags is `purple` composed of? (Python 3.11+)

Rodrigo 🐍🚀
mathspp.com/drops

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.

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

```python
from typing import Literal

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

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

`Literal` side-benefit: documents the valid values.

Rodrigo 🐍🚀
mathspp.com/drops

Return value of a generator

When a generator is finished, you can return an arbitrary value (like a normal function).

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

```python
gen = my_generator_function()
print(next(gen), next(gen))  # 1 2
try:
    next(gen)
except StopIteration as err:
    print(err.value)  # 73
```

The return value of a generator is attached to the exception `StopIteration` it raises when it's finished.

Rodrigo 🐍🚀
mathspp.com/drops

Enumerations of string values

When creating various string values, for example for function options or configurations, you can group them under a string enumeration `StrEnum` from the module `enum`.

```python
from enum import StrEnum

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

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


move(Direction.UP)  # Going up.
move("DOWN")  # Going down. (Doesn't type-check.)
```

If needed, you can use the string directly at runtime.
(But it won't pass type checking.)

⚠️ Python 3.11+ only.

Rodrigo 🐍🚀
mathspp.com/drops

Most recently-modified file

Use the built-in `max` and a custom key function to find the most recently modified file in a folder in a single line of code.

```python
from pathlib import Path

folder = 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
```

`.stat()` provides access to file statistics and `st_mtime` is the modified time.

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

Files only (no directories).

Rodrigo 🐍🚀
mathspp.com/drops

Normalise strings by removing accents

You can use the built-in module `unicodedata` to write a function that removes accents from strings, normalising all characters into the ASCII range.

```python
import unicodedata

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

print(remove_accents("Rodrigo Girão Serrão"))  # Rodrigo Girao Serrao
print(remove_accents("äáàãñç"))  # aaaanc
```

Separates the accents from the letters.

```python
list(unicodedata.normalize("NFD", "äáàãñç"))
# ['a', '̈', 'a', '́', 'a', '̀', 'a', '̃', 'n', '̃', 'c', '̧']
```

Rodrigo 🐍🚀
mathspp.com/drops

Transpose a list of lists

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

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

firsts, lasts = zip(*persons)
print(firsts)
# ('Han', 'Obi-Wan', 'Darth')
print(lasts)
# ('Solo', 'Kenobi', 'Vader')
```

You can also look at this operation as “undoing” what a `zip` does (you can get `firsts` and `lasts` from `persons` and vice-versa).

```
print(
    list(zip(firsts, lasts)) == persons
)  # True
```

Rodrigo 🐍🚀
mathspp.com/drops

Inline lists and tuples

The splat operator * can be used to inline lists, tuples, and other iterables.

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

def more_firsts():
    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']
```

The splat operator * “flattens” the iterables as if their values were written out explicitly inside this list.

Rodrigo 🐍🚀
mathspp.com/drops

Typing iterables instead of lists

If you're accepting an iterable argument, don't type it as a list.
Instead, use `typing.Iterable` or `collections.abc.Iterable`.

```
# ❌
def create_files(files: list[Path]) -> None:
    for file in files:
        ...
```

Using `list[Path]` is too restrictive; all you need is to be able to iterate over `files`.

Python 3.9+

```
from collections.abc import Iterable

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

Python 3.8

```
from typing import Iterable

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

Rodrigo 🐍🚀
mathspp.com/drops

Multi-dictionary

You can create a dictionary that maps a single key to multiple values. (Kind of). The easiest way is with `collections.defaultdict` and the built-in `list`.

```
from collections import defaultdict

multidict = defaultdict(list)
```

All keys will map to an empty list by default:

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

By accessing the key and appending, you map a single key to “multiple” values.

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

Rodrigo 🐍🚀
mathspp.com/drops

Global enumeration members

```python
from enum import Flag, auto, global_enum

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

Use the decorator `global_enum` from the module `enum` to automatically export enumeration members to your globals.

```python
BASE_PERMISSIONS = READ | WRITE
```

```python
BASE_PERMISSIONS = FilePermissions.READ | FilePermissions.WRITE
```

You can still access the members through the enumeration class.

(Class decorator; pretty cool, right?)

Rodrigo 🐍🚀
mathspp.com/drops

Automatic enumeration values

The module enum lets you use auto to automatically assign values to the enumeration members. Works well with flags and string enumerations, too!

```python
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 flags, values are consecutive powers of 2.

```python
from enum import StrEnum, auto

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

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

For string enumerations, values are the lowercase member name.

Rodrigo 🐍🚀
mathspp.com/drops

OS-agnostic line splitting

Strings have a method `splitlines` that you should use when splitting a string on newlines. Don't use .split("\n") because of carriage returns!

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

Lines on Windows can be terminated with `\r\n` instead of just `\n`:

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

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

💡 use `.splitlines(keepends=True)` to preserve the line-ending characters.

Rodrigo 🐍🚀
mathspp.com/drops

Longest and shortest

The built-ins `max`/`min`, the built-in `len`, and `functools.partial`, can be put together to create the custom functions `longest` and `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
```

`partial` lets you create different functions by combining simple functions you already know.

Rodrigo 🐍🚀
mathspp.com/drops

Bounded cache

Use `functools.lru_cache` to create a memory-bounded cache.
The argument specifies how many values to cache.

```python
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)
```

Cache up to 1024 values.

Access cache stats through `.cache_info` on the cached function.

💡 When the cache is full, the Least-Recently Used values (LRU) are evicted to make room for the new ones.

Rodrigo 🐍🚀
mathspp.com/drops

Read files in chunks

The built-in `iter` has the superpower of turning functions into iterables. `iter(f, value)` creates an iterator that calls `f` until it returns the given value.

In this example, iter creates an iterable that reads the file in chunks of 16 characters.

```python
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
...
"""
```

Rodrigo 🐍🚀
mathspp.com/drops

Format specifier `!r`

Use the format specifier `!r` in f-strings to improve your print-debugging.
This uses a value's `repr`, which is better for debugging.

```python
string = "3"
number = 3

print(f"{string} {number}")  # 3 3
print(f"{string!r} {number!r}")  # '3' 3

from fractions import Fraction

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

Strings and numbers look the same... Unless you use `!r`.

Printing a fraction will pretty-print it. Using `!r`, we can be sure it's a `Fraction` object.

Rodrigo 🐍🚀
mathspp.com/drops

Counting values

Do you want to count how many values satisfy a given condition?
Use the built-in sum and a generator expression.

1. The generator expression goes over all values you want to consider...

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

2. The predicate evaluates the condition on each value, returning `True` or `False`.

3. The built-in `sum` accumulates all the Boolean values, effectively counting the number of `True`s.

Rodrigo 🐍🚀
mathspp.com/drops

The evolution of the dot product

How the idiom to compute the dot product has evolved in Python, using the module `operator`, the built-ins `map` and `zip`, and more.

"Standard" idiom in older versions of Python.

```
# Python <=3.11
sum(map(operator.mul, vec1, vec2))
```

```
# Python 3.12+
from itertools import starmap

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

`strict=True` ensures the vectors have the same length.
`itertools.starmap` gets the multiplication "inside" the pairs produced by `zip`.

```
# Python 3.14+
sum(map(operator.mul, vec1, vec2, strict=True))
```

In 3.14 we go back to the original idiom because `map` now supports `strict=True`.

Rodrigo 🐍🚀
mathspp.com/drops

Batching API calls

Since Python 3.12, the module `itertools` provides `batched`, that slices the given iterable in batches of the specified size. Use it for batch processing anything, e.g., API calls:

Hit the API once per user... ❌

```
all_user_data = []
for user in users_to_fetch:
    user_data = api.get_user(user)
    all_user_data.append(
        user_data.stuff_you_want
    )
```

Ask for data from 50 users at a time. ✅

```
from itertools import batched

all_user_data = []
for batch in batched(users_to_fetch, 50):
    users_data = api.get_users(batch)
    all_user_data.extend(
        ud.stuff_you_want
        for ud in users_data
    )
```

Fewer API calls is good because:

- much faster
- less rate limiting

Rodrigo 🐍🚀
mathspp.com/drops

Redact email addresses

You can use f-strings to create the effect of redacted/private data such as phone numbers or email addresses. (The format spec is doing all the work!)

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

Show first two chars.

Align the two chars on the left...

and use *...

to fill the field with the width of the user.

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

Rodrigo 🐍🚀
mathspp.com/drops

Random choices

Use the functions `random.choices` and `random.sample` to randomly pick elements from a sequence, with and without replacement, respectively.

```python
import random

coin_sides = ["heads", "tails"]

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

```python
import random

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

k determines how many values to pick.

💡 Don't use the module `random` for security-sensitive randomness. E.g., don't generate passwords with it. Use the module `secrets` instead.

Rodrigo 🐍🚀
mathspp.com/drops

Dynamic regex replacements

Regex replacements can be computed dynamically, based on the match found. The function `re.sub` accepts a function that accepts the match and returns the replacement.

The function is used instead of a replacement string.

```python
import re

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

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.
```

Rodrigo 🐍🚀
mathspp.com/drops

String constants

The module string defines many useful constants that you can, and should, use!
These will save you from plenty of easily-avoidable bugs.

```python
import string

print(string.ascii_lowercase)
# abcdefghijklmnopqrstuvwxyz

print(string.digits)
# 0123456789

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

These are just 3 of the many available constants!

💡 Think this is silly? I found +10,000 repos on GitHub with bugs on alphabet strings; avoidable with `string.ascii_lowercase`...

Rodrigo 🐍🚀
mathspp.com/drops

Case-insensitive regular expressions

Regular expressions can start with the flag `(?i)`, making them case-insensitive.

```python
import re

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

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

More convenient than lowercasing the strings to check.

Rodrigo 🐍🚀
mathspp.com/drops

Module `itertools` categorisation

The 19 iterables from the module itertools can be classified into 5 categories.
This makes it easier to remember what tools you have at hand.

1

```
# Filtering
from itertools import (
    compress,
    dropwhile,
    filterfalse,
    takewhile,
)
```

2

```
# Reshaping
from itertools import (
    batched,
    chain,
    groupby,
    islice,
    pairwise,
)
```

3

```
# Tool-complementing
from itertools import (
    accumulate,    # reduce
    starmap,       # map
    zip_longest,   # zip
)
```

4

```
# Infinite
from itertools import (
    count,
    cycle,
    repeat,
)
```

5

```
# Combinatorial
from itertools import (
    combinations,
    combinations_with_replacement,
    permutations,
    product,
)
```

💡 There's also `tee`, which is not an iterable.

Rodrigo 🐍🚀
mathspp.com/drops

t-strings for safe string formatting

t-strings (T, not F) are composed of string segments and the interpolated values. The interpolated values must be processed/converted by you, which means you can sanitise them.

```
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>
#       ^^^^     ^^^^                        ^^^^     ^^^^
```

The potentially malicious HTML, that contained a JS script, was escaped.

```
from string.templatelib import Template

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

💡 Python 3.14+ only

Rodrigo 🐍🚀
mathspp.com/drops

Structural unpacking

Python iterables (lists, tuples, ...) can be unpacked according to their length and nesting.

```
colour_info = ("AliceBlue", (240, 248, 255, 255))
#                           name   red green blue alpha

name, (*rgb, alpha) = colour_info
```

The splat operator `*` can be used at any level of nesting to capture 0 or more values.

Tuples on the left of the assignment capture nesting.

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

💡 Structural unpacking also works in `for` loops.

Rodrigo 🐍🚀
mathspp.com/drops

Ergonomic multiline strings

When writing multiline strings I like to have the """ on their one lines, which adds extra newlines at the beginning and end of the string. Use \ to escape them.

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

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

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

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

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

Avoid the leading newline.

Avoid the trailing newline.

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

Rodrigo 🐍🚀
mathspp.com/drops

Underscore in the REPL

In the REPL, the special value _ refers to the result of the most recent non-None expression.

The result is saved in _.

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

>>> _
7625597484987

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

The function print returns None, so _ still holds the previous result.

The function sum produces a non-None value, so that updates the value in _.

Rodrigo 🐍🚀
mathspp.com/drops

Subclassing immutable types

If you need to subclass an immutable type (uncommon, but not unheard of), you will need to override `__new__`, which runs before `__init__`.

Use the value (`4.5`) to create the `float` part.

```
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
```

Subclass-specific customisation typically goes in `__init__`.

Rodrigo 🐍🚀
mathspp.com/drops

💡 There's more to the story than this, but this is the

Idiomatic sequence slicing

The five sequence slicing patterns below are idioms you must learn so that you can understand the patterns automatically, without thinking.

```
string = "Slicing is easy!"
```

`[:n]` – first n chars.

```
print(slicing[:4])
# Slic
```

`[:-n]` – without last n chars.

```
print(slicing[:-4])
# Slicing is e
```

`[::-1]` – reversed.

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

Built-in `reversed` is usually preferred.

`[n:]` – after first n chars.

```
print(slicing[4:])
# ing is easy!
```

`[-n:]` – last n chars.

```
print(slicing[-4:])
# asy!
```

Rodrigo 🐍🚀
mathspp.com/drops

File tail

To get a file tail (the last few lines of a file), use a `deque` from the module `collections`.

```python
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]))
```

Number of last lines to fetch.

💡 If the file has fewer lines than those requested, you get the full file.

Rodrigo 🐍🚀
mathspp.com/drops

One-shot file I/O

For one-shot file I/O, you can use the methods `write_text` and `read_text` from `pathlib.Path`.

```python
from pathlib import Path

filepath = Path("hello_world.txt")

filepath.write_text("Python is cool!")

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

If the file exists, `write_text` overwrites the contents of the file.

You get more control if you use a context manager and `open` but this gets the job done in most cases.

Rodrigo 🐍🚀
mathspp.com/drops

Formatting big numbers

To improve readability when printing big numbers, use format specifiers to include thousands separators.

```
bignum = 123541241234

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

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

print(f"Big money ${x:n}")
# Big money $123541241234
```

The comma (,) uses commas as the thousands separator.

The underscore (_) uses underscores as the thousands separator.

The specifier n uses a locale-appropriate thousands separator...

... which may be none!

Rodrigo 🐍🚀
mathspp.com/drops

Named groups in regex

When using regex with Python's module re, you can use the syntax `(?P<...>)` to create a named group, making it easier to retrieve match groups.

The group name goes inside the <>.

```python
import re

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

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

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

💡 This is a Python-specific regex feature.

Rodrigo 🐍🚀
mathspp.com/drops

Resolving paths

When working with paths, use the methods `expanduser` and `resolve` from `pathlib.Path` to get expanded paths that are absolute.

```
from pathlib import Path

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

downloads = docs / "..." / "Downloads"
print(downloads)  # ~/Documents/../Downloads
print(downloads.expanduser().resolve())  # /Users/rodrigogs/Downloads
```

The reverse order gives the wrong result...

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

Rodrigo 🐍🚀
mathspp.com/drops

Formatting dates in f-strings

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

```python
from datetime import date

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

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

The “format specifier” after the colon : can contain arbitrary characters and only the known specifiers are replaced.

Rodrigo 🐍🚀
mathspp.com/drops

Concatenate files from handlers

If you want to concatenate multiple files and you ***already*** have a list of file handlers, use `itertools.chain.from_iterable`.

```
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
```

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

from itertools import chain

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

💡 If you don't have the files open yet, it's best to open one at a time.

```
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
```

Rodrigo 🐍🚀
mathspp.com/drops

Generator recipe

To write a generator, first write a function that returns the list with the elements you want. Then, convert to a generator function.

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

Yield instead of appending.

```python
def squares(stop):

    for n in range(stop):
        yield n ** 2
```

Same values produced.

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

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

💡 The trick also works for infinite generators, even if the list version never returns.

Rodrigo 🐍🚀
mathspp.com/drops

Partition a string in two halves

To split a string over a separator once, use the method `str.partition`. This returns a 3-item tuple with the results of the split. The middle item can be used to check if the split was successful.

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

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

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

if not at:
    print(user)  # mathspp.com
```

The 2nd and 3rd items are empty if no split happened.

```
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')
```

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

Rodrigo 🐍🚀
mathspp.com/drops

Regex multiline flag

The multiline flag `(?m)` makes it so that the special characters ^ and $ match the beginning & end of each newline instead of the beginning & end of the string.

```
the = r"^[Tt]he"
```

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

The caret ^ matches the beginning of the string.

```
the = r"(?m)^[Tt]he"
```

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

The caret ^ matches the beginning of each new line because of the multiline flag `(?m)`.

Rodrigo 🐍🚀
mathspp.com/drops

Extract assignments from conditionals

Instead of using a conditional statement to make an assignment, use a conditional expression.

“If the user is logged in,
set their permissions to "full",
otherwise
set them to "guest".”

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

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

“Set the user permissions to "full"
if the user is logged in,
otherwise set them to "guest".”

Conditional expressions are very effective if you learn to read them!

Rodrigo 🐍🚀
mathspp.com/drops

File discovery by name pattern

If you want to find all files in a directory that have a name that follows a given pattern, you can use the method `glob` from `pathlib.Path`.

Find all files with “cat” in the name.

```python
from pathlib import Path

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

Find all files that end with “.jpg”.

```python
from pathlib import Path

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

Filesystem:

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

💡 The order of the files is arbitrary.

Rodrigo 🐍🚀
mathspp.com/drops

Structural pattern matching with dictionaries

Structural pattern matching can be used to match dictionaries and their keys/values.

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

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

✅ matches even though there are extra key/value pairs.

❌ Missing key "age".

```
my_dict = {"user": "John"}
```

❌ "age" should be an integer but it's a string.

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

Rodrigo 🐍🚀
mathspp.com/drops

Slicing generators for debugging

When debugging generators, `itertools.islice` can be used to slice a generator to get the first few elements for inspection.

```
from itertools import islice

gen = mysterious_generator()

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

You can use `list(islice(..., n))` to get the first n elements immediately.

Don't forget that subsequent usages of `islice` will slice from the *remainder* of the generator.

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

Rodrigo 🐍🚀
mathspp.com/drops

Peek at an iterable

To “peek” at an iterable you fetch its next element and then “glue” it back with `chain` from the module `itertools`.

```python
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.

Rodrigo 🐍🚀
mathspp.com/drops

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__`, a tuple of strings.

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

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

The tuple must specify the attribute names used for matching. If it's not listed, it's ignored when matching.

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

The pattern with `Point` only matches the attributes x and y, ignoring z.

Rodrigo 🐍🚀
mathspp.com/drops

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.

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

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

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

```
print(next(squares, "✨"))
# ✨
```

A default value (of `None`, for example) tends to be more useful in idioms that use `next`.

Rodrigo 🐍🚀
mathspp.com/drops

Match an exact dictionary structure

Structural pattern matching can be used to match dictionaries with given keys and values.
By default, dictionaries with extra keys match, but this can be disabled with a guard.

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

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

```python
{}  # ❌
{"name": 73}  # ❌
{"name": "John"}  # ✅
{"name": "John", "age": 42}  # ✅
```

```python
{}  # ❌
{"name": 73}  # ❌
{"name": "John"}  # ✅
{"name": "John", "age": 42}  # ❌
```

The guard “`not kwargs`” ensures there are no extra key/value pairs.

Rodrigo 🐍🚀
mathspp.com/drops

Undoable iterator with value history

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

```
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
```

History of seen values.

Queue for values that were "undone" and that are the next ones to be produced (again).

Next value? Ensure there's a queued value and get that.

Undo? Move from the history to the queue.

Rodrigo 🐍🚀
mathspp.com/drops

Typing overloads

Use typing overloads to specify explicit relationships between the types of arguments and the return type of your functions.

Type checkers say the return type is `str | int`, even though we know it's `str`.

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

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

```
from typing import overload

@overload
def double(arg: str) -> str: ...

@overload
def double(arg: int) -> int: ...

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

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

The overloads allow you to specify relationships between the various argument types and the return type.

Type checkers use that information.

Rodrigo 🐍🚀
mathspp.com/drops

Bulk renaming files

To change the name of a file while preserving the same extension you can use the method `with_stem` from `pathlib.PurePath`. Useful for bulk renaming operations.

```python
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
# ...
```

Rodrigo 🐍🚀
mathspp.com/drops

Structural validation and homogenisation

The `match` statement can be used to validate and homogenise structures. The example below mimics CSS padding, where the user can specify values in different formats.

```python
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}.")
```

```python
# ✅
unpack_padding(10)
unpack_padding((5, 10))
unpack_padding((1, 2, 3, 4))
```

```python
# ❌
unpack_padding("hello")
unpack_padding(("5", "10"))
unpack_padding((5, 10, 15))
```

Rodrigo 🐍🚀
mathspp.com/drops

Verbose regular expressions

Possibly the best thing you can do for yourself when writing a regular expression is to turn on the verbose mode, allowing you to comment and split your regex across multiple lines.

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

Same regex.

“Verbose” flag.

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

Comments inside the regex.

Whitespace is ignored, namely spaces and newlines.

To match spaces escape them with the backslash \.

Rodrigo 🐍🚀
mathspp.com/drops

Extracting text data into a dict

When using regular expressions to extract data from structured text, named groups allow you to use the method `re.Match.groupdict` to automatically build a dict with the results.

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

# 1 - Loomings

Call me Ishmael. [...]
```

```
import re

re.match(pattern, text).groupdict()
```

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

```
{'title': 'Moby Dick',
 'author': 'Herman Melville'}
```

Named groups turn into key/value pairs.

Rodrigo 🐍🚀
mathspp.com/drops

Generics syntax

In Python 3.12 a new syntax for generics (functions and classes) was introduced, streamlining the process of defining generics. Use `[...]` to indicate the type variables after the object name.

The `[T]` introduces the type variable T, which can be used in other annotations. It specifies relationships between types.

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

```python
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()
```

The values returned from `pop` and the values accepted by `append` have the same type.

Rodrigo 🐍🚀
mathspp.com/drops

Add lists together, fast

Adding (concatenating) lists is a “slow” operation because it requires creating a new list with the items from both other lists. In performance-sensitive scenarios, chain them instead.

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

Suppose you need to traverse a tree structure, starting at “`this_node`” and going up its parents.

```
from itertools import chain

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

`itertools.chain` is faster:

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

Rodrigo 🐍🚀
mathspp.com/drops

File modes

When opening a file, you can specify a mode. This determines what operations are possible. **r** for reading, **w** for writing, **a** for appending, and **x** for exclusive creation of the file.

```
with open("path/to/file.txt", mode="..."):
    pass
```

The default mode is **r**.

Open the file for writing while ensuring the file didn't exist yet.

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

💡 Add **+** to the file mode to allow writing & reading at the same time.

💡 Add **b** to the file mode to open in binary mode. Default is text mode (**t**).

Rodrigo 🐍🚀
mathspp.com/drops

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.

```
from functools import cache

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

Sets aren't hashable, so caching won't work.

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

Frozen sets are immutable versions of sets and hashable.

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

Rodrigo 🐍🚀
mathspp.com/drops

Constrained generics

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

There are no restrictions on what T can be:

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

The variable type T must be one of the given types:

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

```
double(None)
# This gives an error at runtime.
```

In the constrained, version the type checker detects the error.

In the unconstrained version, the type checker doesn't detect the error and thinks `double(None)` will return None.

Rodrigo 🐍🚀
mathspp.com/drops

Natural alphabetical sorting

Strings are sorted lexicographically based on the characters' codepoints, which means upper-case and lower-case letters aren't sorted together. Use `str.casefold` to fix that.

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

Upper-case letters come before lower-case letters in Unicode, hence they're sorted first.

Using `key=str.casefold` effectively ignores casing when sorting, so you get the natural order A -> B -> C.

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

Rodrigo 🐍🚀
mathspp.com/drops

Preserving decorated function metadata

The standard approach to writing decorators has the drawback of “erasing” function metadata. You can fix this by using the decorator `functools.wraps` in your decorator (how meta!).

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

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

print(add)  # <function my_decorator.<lo...
help(add)  # Cryptic help message.
```

```python
import functools

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

# ...

print(add)  # <function add at 0x10575c510>
help(add)  # Fixed signature & docstring.
```

By default, the name of the function gets messed up, and so does the docstring used when getting help.

Rodrigo 🐍🚀
mathspp.com/drops

Timestamp file names

To add a timestamp to a file name you can use the module `datetime`. `datetime.datetime.now` creates a representation of the current time and then string formatting turns it into a timestamp.

```python
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 timestamp contains the year, month, day, hour, minutes, and seconds.

Add `%f` to go down to the microsecond.

Rodrigo 🐍🚀
mathspp.com/drops

Temporary directory

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.

```python
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 done, the directory and its contents are removed.

You can create & manipulate “normal” files here, as they'll get cleaned up automatically.

Rodrigo 🐍🚀
mathspp.com/drops

Non-local variables

To access non-local variables in closures, use the keyword `nonlocal`. Useful for decorators.

`counter` is defined outside `inner` but inside `outer`. It's not local to `inner`, nor is it global. It's non-local.

```
def outer():
    counter = 0
    def inner():
        nonlocal counter
        counter += 1
        print(counter)
    return inner

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

To be able to write to the variable, it must be "declared" as non-local.

If you only wanted to read the value, `nonlocal` wouldn't be needed.

💡 If the value is mutable (like a list or a dictionary), you can mutate it without using `nonlocal`.

Rodrigo 🐍🚀
mathspp.com/drops

Dynamic width string formatting

String formatting accepts formatting specifiers dynamically. This is useful, for example, to compute the maximum width of a column from a list of strings.

```python
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}")
```

All fruits are left-aligned on a field with a width computed dynamically.

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

An extra pair of curly braces { } tells Python to use the value of the variable as the formatting, here using 9 as the width specifier.

Rodrigo 🐍🚀
mathspp.com/drops

Docstring attribute

Docstrings are saved as the attribute `__doc__` on the objects they were defined in.
This is a readable and writable string.

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

```
fn.__doc__ = "Bye!"
```

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

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

Rodrigo 🐍🚀
mathspp.com/drops

Common __hash__ implementation

The dunder method __hash__ can be implemented to make your instances hashable. The most common approach is to hash a tuple with all the attributes that are “important”.

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

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

```
f = Fraction(1, 3)

print(hash(f))
# -1440771752368011620
```

The “important” attributes are the ones that you'd use when comparing two instances to check if they're equal.

Rodrigo 🐍🚀
mathspp.com/drops

Match word boundaries

The special character \b can be used in regular expressions to match word boundaries. The special character \B matches inside words.

```
pattern = r"water"
text = "watermelon is 92% water!"
```

Matches the five characters "water".

```
pattern = r"\bwater\b"
text = "watermelon is 92% water!"
```

Matches "water" as a standalone word.

```
pattern = r"\bwater\B"
text = "watermelon is 92% water!"
```

Matches "water" as a prefix.

Combining \b and \B allows you to match standalone words, prefixes, suffixes, ...

Rodrigo 🐍🚀
mathspp.com/drops

Custom containment checks

The dunder method __contains__ can be used to implement containment checks with the keywords `in` and `not in`.

Represents an interval of all numbers between the left and right points, inclusive.

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

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

```
interval = ClosedInterval(2.7, 4.5)

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

Containment checks happen by calling the method `__contains__`, which must return a Boolean value.

Rodrigo 🐍🚀
mathspp.com/drops

Readable object names

Objects like functions and classes have a dunder attribute `__name__` that gives you access to a readable version of that object's name. Useful, for example, in a class's `__repr__`.

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

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

```
print(Point(3))  # Point(3)
```

Using `__name__` is robust to changes to the class name and works better with inheritance.

Functions also have a `__name__`:

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

Rodrigo 🐍🚀
mathspp.com/drops

Filtering Truthy values

The built-in `filter` accepts the special argument `None` instead of a predicate function. In that case, `filter` will remove all Falsy values from the given iterable.

```python
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.
```

`filter(None, ...)` skips all empty strings and `None` because those values are Falsy.

Rodrigo 🐍🚀
mathspp.com/drops

Pretty-printing

The standard library offers a module `pprint` that contains functionality for pretty-printing data structures. The function `pprint.pprint` is the pretty-printing equivalent to the built-in `print`.

```python
from pprint import pprint
pprint(globals())
```

The built-in `print` will cram everything into one line, while `pprint` will spread its output across lines.

```python
print(globals())
```

```
{'__builtins__': <module 'builtins' (built-in)>,
 '__doc__': None,
 '__loader__': <_frozen_importlib_external.SourceFileLoade
 '__name__': '__main__',
 '__package__': None,
 '__spec__': None,
 'partial': <class 'functools.partial'>,
 'pprint': <function pprint at 0x10362c1a0>}
```

```
{'__name__': '__main__', '__doc__': None, '__package__'
: None, '__loader__': <_frozen_importlib_external
.SourceFileLoader object at 0x103121910>, '__spec__':
None, '__builtins__': <module 'builtins' (built-in)>}
```

💡 `pprint` is particularly useful when pretty-printing nested data structures.

Rodrigo 🐍🚀
mathspp.com/drops

__str__ vs __repr__

The dunder methods `__str__` and `__repr__` both serve to create a string representation of an object. Use `__str__` for a "pretty" representation and `__repr__` for a "debug" representation.

```python
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})"

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

`__repr__` is more important than `__str__`. When in doubt, implement only `__repr__`.

Rule of thumb: `__repr__` should produce a string that can be copied & pasted to recreate the object.

Rodrigo 🐍🚀
mathspp.com/drops

Typing *args and **kwargs

When typing `*args` and `**kwargs`, the type hints should target the values that will be passed as positional or keyword arguments, and not the tuple `*args` or the dictionary `**kwargs`.

```
my_max(73, 42, 10, reversed=True)
```

❌

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

This is a function that accepts an arbitrary number of tuples as positional arguments and an arbitrary number of dictionaries as keyword arguments.

✅

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

Rodrigo 🐍🚀
mathspp.com/drops

AST parsing

The module `ast` has a function `parse` that lets you parse code in a string into an Abstract Syntax Tree (AST), an intermediate representation of your code that Python operates on.

```
if a:
    pass
elif b:
    pass
else:
    pass
```

```
if a:
    pass
else:
    if b:
        pass
    else:
        pass
```

```python
import ast

code = """..."""

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()])])])
```

The two pieces of code produce the same AST, which shows that `elif` and an `if` immediately after an `else` are the same thing for Python.

Rodrigo 🐍🚀
mathspp.com/drops

Dunder method `__missing__`

The dunder method `__missing__` is used by `dict` subclasses when you look up a key that isn't in the dictionary. You could use it to reimplement `collections.defaultdict`, for example.

```python
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]
```

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

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

`__missing__` is called after `__getitem__` checks the key isn't there. For key normalisation before key look-ups, overriding `__getitem__` is better.

Rodrigo 🐍🚀
mathspp.com/drops

Custom enum search behaviour

Akin to the dunder method `__missing__`, enums support a classmethod `_missing_` (single underscore) that can be used to implement custom enum search behaviour.

```python
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
```

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

UK/US-friendly enumerations.

`_missing_` is called when instantiating an enum with a value that's not legal.

Rodrigo 🐍🚀
mathspp.com/drops

map's keyword argument `strict`

The built-in function `map` has a keyword argument `strict` that you can use to signal that the 2+ argument iterables should have the same length. If they don't, a `ValueError` is raised.

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

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

By default, the built-in `map` stops at the end of the shorter iterable.

```
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
```

By setting `strict=True`, you get a `ValueError` when you hit the end of the shorter iterable.

💡 Python 3.14+ only

Rodrigo 🐍🚀
mathspp.com/drops

Dynamic module attribute look-up

Modules may define the dunder method `__getattr__`, which may be used to load names dynamically. This is helpful, for example, to lazy-load heavy objects / to keep import times short.

my_module.py

```python
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" +
        f" no attribute {name!r}"
    )
```

Accessing the name `HeavyClass` triggers the lazy importing.

```python
import my_module

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

obj2 = my_module.HeavyClass()
```

Subsequent accesses don't trigger `__getattr__` because the name is globally available in the module.

Rodrigo 🐍🚀
mathspp.com/drops

Add typing to decorators

Decorators and other higher-order functions can use `typing.ParamSpec` to annotate the `*args` and `**kwargs` of the inner/wrapper functions that they define.

We'll use this no-op decorator as an example:

```
def decorator(f):
    def wrapper(*args, **kwargs):
        return f(*args, **kwargs)
    return wrapper
```

The decorator accepts a function with a given "**param**eter **spec**ification" (arguments) and returns a function with the same parameter specification.

```
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 `.args` and `.kwargs` attributes of the `ParamSpec` link the original types of the arguments of `f` to the types of the arguments of `wrapper`.

In 3.12+, the `**P` uses the new generic syntax to create a `ParamSpec`.

Rodrigo 🐍🚀
mathspp.com/drops

Non-greedy regex quantifiers

In regular expressions, use the question mark ? after a quantifier (*, +, and ?) to make that quantifier non-greedy, meaning it will try to match as few characters as possible.

```
import re

html = "<a href='mathspp.com'>My site</a>"
```

Note how there are two > characters in the string.

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

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

The greedy pattern `.*` will match as many characters as possible before matching >.

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

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

The non-greedy pattern `.*?` will only match what's absolutely required before matching >.

Rodrigo 🐍🚀
mathspp.com/drops

Type hints that refer to functions

Use `collections.abc.Callable` to add type hints to arguments/return types that are functions themselves. This is useful for higher-order functions.

```
from collections.abc import Callable

Callable[[], bool]
Callable[[float, float], float]
```

The first value is a list with the types of the arguments.

The second value is the return type of the function.

An empty list means the function takes no arguments.

```
def f() -> bool:
    return True
```

```
def add(a: float, b: float) -> float:
    return a + b
```

Rodrigo 🐍🚀
mathspp.com/drops

Two-dimensional range

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

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

In `product`, the first argument is traversed "slowly" while the second argument is traversed over and over:

```python
print("x y")
for x, y in range_2d(2, 3):
    print(x, y)
```

```
x y
0 0
0 1
0 2
1 0
1 1
1 2
```

Rodrigo 🐍🚀
mathspp.com/drops

Constant variables

How do you enforce constant variables? You don't. But using `ALL_UPPERCASE` variable names is a very common convention and you can also use `typing.Final` at the type hint level.

```
PI = 3.14159265
```

A variable name all in uppercase denotes a (global) constant that shouldn't be changed.

```
# ...

PI = 2
# Although upsetting,
# doesn't raise
# an error.
```

None of these enforce anything at runtime and the variable can still be reassigned to.

```
from typing import Final

PI: Final[float] = 3.14159265
```

The type hint `Final` tells static type checkers that a variable shouldn't be reassigned to.

The actual type of the variable goes inside the square brackets `[...]`.

Rodrigo 🐍🚀
mathspp.com/drops

Initialise a dictionary from its keys

If you have an iterable of keys, you can initialise a dictionary with them by using the class method `dict.fromkeys`, which creates a dictionary where all values are set to `None`.

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

print(person_info)
```

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

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

print(person_info)
```

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

The value can be specified as the second argument.
(Careful with mutable values.)

Rodrigo 🐍🚀
mathspp.com/drops

Name mangling

Inside a class, attributes or methods whose name has a leading double underscore are mangled so it's harder to overwrite by mistake those attributes/methods when subclassing.

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

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

Other methods inside the same class can see the attributes/methods with their original names.

```python
c = C()
c.print_value()
# 42
```

```python
print(c.__value)  # AttributeError
print(c._C__value)  # 42
```

Outside the class, attributes/methods can only be accessed through their mangled name.

Rodrigo 🐍🚀
mathspp.com/drops

Send data into generators

Generators have a method `send` that you can use to send information into a generator. Inside the generator, the data is the result of the expression `yield`.

```python
def count():
    at = 0
    while True:
        skip = yield at
        if skip is not None:
            at += skip
        else:
            at += 1
```

```python
counter = count()
print(next(counter))  # 0
print(next(counter))  # 1
```

Values sent are retrieved from the `yield` expression.

```python
print(counter.send(5))  # 6

print(next(counter))  # 7
```

Sending a value also produces the next result.

When using the built-in `next`, the result of the `yield` is `None`. (Because no value was sent.)

Rodrigo 🐍🚀
mathspp.com/drops

Dictionary creation idiom

Dictionaries can be created from iterables of key/value pairs, which gives rise to a useful idiom to create dictionaries using the built-ins `dict` and `zip`.

```python
prompts = ["age", "country", "favourite language"]
answers = [28, "Portugal", "Python"]

info = dict(zip(prompts, answers))
```

Keys are taken from the first iterable and paired with the value from the second iterable in the corresponding position.

```python
{
    'age': 28,
    'country': 'Portugal',
    'favourite language': 'Python',
}
```

Rodrigo 🐍🚀
mathspp.com/drops

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.

```python
class Animal:
    def sound(self) -> str:
        ...

    def sleep(self) -> None:
        ...
```

```python
from typing import overrides

class Cat(Animal):
    @overrides
    def sound(self) -> str:
        ...

    @overrides
    def seep(self) -> None
        ...
```

OK!

"`seep`" is misspelled, so it isn't actually overriding anything. Because of that, type checkers will complain and you can fix the typo.

💡 More than catching typos, `overrides` will "warn" you if method names in the parent class(es) change, e.g., after upgrading package versions.

Rodrigo 🐍🚀
mathspp.com/drops

Efficiently extending a list

The method `extend` is the efficient way of adding multiple elements to a list in one go.

```
my_list = [...]

for value in other_list:
    my_list.append(value)
```

Successively appending will waste time when constantly resizing the list.

```
my_list = [...]

my_list += other_list
```

Concatenating the two lists wastes time and memory temporarily creating a third list.

```
my_list = [...]

my_list.extend(other_list)
```

The method `extend` is efficient and also works with other (non-list) iterables.

Rodrigo 🐍🚀
mathspp.com/drops

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.

```python
from itertools import groupby

def all_equal(iterable, *, key=None):
    return len(take(
        groupby(iterable, key),
        2,
    )) <= 1
```

```python
from itertools import islice

def take(iterable, n):
    return list(islice(iterable, n))
```

Auxiliary function that takes the first n elements from an iterable as a list.

`groupby` creates "runs" of elements that are the same under the function `key`.
If we only have 0 or 1 unique groups, then all elements are equal under the function `key`.

```python
# Example:
all_equal([True, 1, "1", 1.0], key=int)
# True
```

Rodrigo 🐍🚀
mathspp.com/drops