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