
Better
PYTHON CODE
David Mertz
Foreword by Alex Martelli

Praise for *Better Python Code*

"You'll not just be aspiring to be an expert anymore after practicing through *Better Python Code: A Guide for Aspiring Experts*, you'll be one of them! Learn from David Mertz, who's been making experts through his writing and training for the past 20 years."
—*Iqbal Abdullah, past Chair, PyCon Asia Pacific, and past board member, PyCon Japan*

"In *Better Python Code: A Guide for Aspiring Experts*, David Mertz serves up bite-sized chapters of Pythonic wisdom in this must-have addition to any serious Python programmer's collection. This book helps bridge the gap from beginner to advanced Python user, but even the most seasoned Python programmer can up their game with Mertz's insight into the ins and outs of Python."
—*Katrina Riehl, President, NumFOCUS*

"What separates ordinary coders from Python experts? It's more than just knowing best practices—it's understanding the benefits and pitfalls of the many aspects of Python, and knowing when and why to choose one approach over another. In this book David draws on his more than 20 years of involvement in the Python ecosystem and his experience as a Python author to make sure that the readers understand both *what* to do and *why* in a wide variety of scenarios."
—*Naomi Ceder, past Chair, Python Software Foundation*

"Like a Pythonic BBC, David Mertz has been informing, entertaining, and educating the Python world for over a quarter of a century, and he continues to do so here in his own pleasantly readable style."
—*Steve Holden, past Chair, Python Software Foundation*

"Being expert means someone with a lot of experience. David's latest book provides some important but common problems that folks generally learn only after spending years of doing and fixing. I think this book will provide a much quicker way to gather those important bits and help many folks across the world to become better."
—*Kushal Das, CPython Core Developer and Director, Python Software Foundation*

"This book is for everyone: from beginners, who want to avoid hard-to-find bugs, all the way to experts looking to write more efficient code. David Mertz has compiled a great set of useful idioms that will make your life as a programmer easier and your users happier."
—*Marc-André Lemburg, past Chair, EuroPython, and past Director, Python Software Foundation*

```
    if not 0 <= index <= 9999:
        raise IndexError(f"Cannot access temporary file {index}")
    with open(filenames[index], mode="a") as tmpfile:   # ❶
        data = get_information_to_store()               # ❷
        tmpfile.write(data)
```

❶ Under the presumption that the same index is repeated, append mode is likely better.

❷ As in other sections, a sample implementation of this function is at https://gnosis.cx/better; any function that returns varying strings is equally illustrative.

3.3.2 Second Danger

The second danger is that failure to close an open file may leave some queued changes unwritten to disk. In fact, even the permissions or existence of an unclosed file could be messed up. Again, using a context manager assures safety around this.

Unsafe open file in `code/crash.py`

```
import os
fh = open("crash.txt", mode="w")
fh.write("Hello, world!\n")
fh.flush()
fh.write("Goodbye!\n")
os._exit(1)
fh.close()
```

Obviously this program is a toy example. Notice, however, that it *has* a `.close()` method call included (which is not reached):

```
[PythonMistakes]$ python code/crash.py         # ❶
[BetterPython]$ cat crash.txt                  # ❷
Hello, world!
```

❶ Run the program.

❷ Show the full output generated within *crash.txt*.

3.3.3 Correcting the Fragility

Simply by enclosing every `open()` within a context manager, the dangers are mitigated.

Safe open file in `code/safe-crash.py`

```
import os
with open("crash.txt", mode="w") as fh:
```

```
    fh.write("Hello, world!\n")
    fh.write("Goodbye!\n")
os._exit(1)
```

Admittedly, a genuine system-level crash such as simulated by `os._exit()` will interfere with flushing unclosed files. That is, if the crash occurred between the "Hello" and "Goodbye" writes, we still wouldn't get all the data to disk. But keeping writes inside the `with` block at least minimizes the exposure to that danger:

```
[PythonMistakes]$ python code/safe-crash.py
[BetterPython]$ cat crash.txt
Hello, world!
Goodbye!
```

Done correctly, all the `fh.write()` lines produce output to *crash.txt*. You can read more about writing your own context managers at https://docs.python.org/3/reference/datamodel.html#context-managers. The excellent description in the Python documentation describes the "guts" of how context managers work internally.

3.4 Optional Argument `key` to `.sort()` and `sorted()`

Using the optional `key` argument to `sorted()` and `list.sort()` can make your code cleaner, faster, and more robust. Failing to use a `key` function where relevant is a common mistake.

The requirements for sorting collections or iterables in Python are suprisingly minimal. Each object must be able to perform a *less than* comparison with the objects adjacent to it in a sequence.

Given that all we need is less-than inequality, providing sortability of custom objects is extremely easy to make available. Here is an example.

Creating custom objects that can be sorted

```
>>> class Thing:
...     def __init__(self, value):
...         self.value = value
...     def __lt__(self, other):
...         return self.value < other.value
...     def __repr__(self):
...         return f"Thing({self.value})"
...
>>> sorted([Thing(2), Thing(-17), Thing(55), Thing(7)])
```

```
[Thing(-17), Thing(2), Thing(7), Thing(55)]
>>> things = [Thing(2), Thing(-17), Thing(55), Thing(7)]
>>> things.sort()
>>> things
[Thing(-17), Thing(2), Thing(7), Thing(55)]
```

Note The pointy edges of sorting

Not all objects can be compared for less-than inequality, which can occasionally have the surprising effect that the sortability of a sequence depends on the original order of elements.

While this can possibly occur, far more often sorting heterogeneous iterables simply fails with some variety of `TypeError`. Still, we *can* see situations like this:

```
>>> sorted([5, Strange(), 1, Strange(), 2+3j])
[1, 5, StrangeObject, StrangeObject, (2+3j)]
>>> sorted([5, Strange(), 2+3j, Strange(), 1])
[1, 5, StrangeObject, StrangeObject, (2+3j)]
>>> sorted([5, Strange(), 1, 2+3j, Strange()])
Traceback (most recent call last):
[...]
TypeError: '<' not supported between instances of
'complex' and 'int'
```

To really understand when this will succeed and when it will fail, for a given sequence of objects that are partially incompatible, you need to understand the details of the Timsort algorithm (https://en.wikipedia.org/wiki/Timsort). Doing so is a worthwhile goal, but is not required to understand anything in this book.

A more useful "thing" would presumably have additional attributes and methods, but this suffices to show the scaffolding needed (the `.__repr__()` is optional, but it makes for a more attractive display).

If a developer is not aware of the optional keyword argument `key`, which can be passed to `sorted()` or to `list.sort()`, the code they write is likely to perform inefficiently or just plain wrongly. In particular, such flawed code can sometimes wind up sorting on a basis other than the sort order that is useful for the objects involved.

For example, suppose we wanted to sort "Things" not based on their numeric order, but rather based on their numeric order within a ring of a given modulus (called $\mathbb{Z}_n$). A first inclination might be to subclass `Thing` to have this behavior.

Note Iterables are sortable (if the corresponding concrete collection is)

Python really does emphasize iterables over concrete collections wherever they are feasible to use. The result of sorting is still a concrete collection, but the iterable need not have started out as such. For example:

```
>>> from random import randint
>>> def make_things():
...     for _ in range(5):
...         yield Thing(randint(1, 1000))           # ❶
...
>>> sorted(make_things())
[Thing(544), Thing(651), Thing(666), Thing(799),
Thing(920)]
```

❶ The presence of `yield` makes this a *generator function*.

Unnecessary use of subclass merely for sort order

```
>>> class ModThing(Thing):
...     def __init__(self, value: int, mod: int=7):
...         self.value = value
...         self._mod = mod
...     def __lt__(self, other):
...         return self.value % self._mod < other.value % other._mod
...
...
>>> sorted([ModThing(2), ModThing(-17), ModThing(55), ModThing(7)])
[Thing(7), Thing(2), Thing(-17), Thing(55)]
```

There might well be additional reasons to attach the modulus to the class itself, but supposing we only cared about sorting, we could achieve the same effect more easily using the following.

Using the `key` function in decorate-sort-undecorate sorting

```
>>> sorted([Thing(2), Thing(-17), Thing(55), Thing(7)],
...        key=lambda thing: thing.value % 7)
[Thing(7), Thing(2), Thing(-17), Thing(55)]
```

Anything that can represent a consistent transformation of the underlying objects being sorted is suitable as a `key` function. The decorate-sort-undecorate pattern is vastly more efficient as a big-O complexity than using a comparison function between every pair of items. See a discussion at https://en.wikipedia.org/wiki/Schwartzian_transform. The less

efficient comparison function idiom is still used in many other programming languages, and was long ago used in Python, prior to version 2.4.

Lambda functions are absolutely appropriate to use in this context, even if in most other places a named function would serve clarity better. But very often it is useful to use `operator.itemgetter` or `operator.attrgetter` as faster and more expressive functions than custom lambda functions. One place we see this need very commonly is in manipulating deserialized JSON data, which tends to be highly nested.

Using `operator.itemgetter` to sort based on a `dict` key

```
>>> from operator import itemgetter
>>> students = [
...     dict(name="Xian", grade="B", age=10),
...     dict(name="Jane", grade="B", age=12),
...     dict(name="John", grade="A", age=15)
... ]
>>> sorted(students, key=itemgetter('age'), reverse=True)
[{'name': 'John', 'grade': 'A', 'age': 15},
 {'name': 'Jane', 'grade': 'B', 'age': 12},
 {'name': 'Xian', 'grade': 'B', 'age': 10}]
>>> sorted(students, key=itemgetter('name'))
[{'name': 'Jane', 'grade': 'B', 'age': 12},
 {'name': 'John', 'grade': 'A', 'age': 15},
 {'name': 'Xian', 'grade': 'B', 'age': 10}]
```

For data held in classes `operator.attrgetter` is very similar, but simply accesses the attribute that is passed as an argument for each instance being sorted.

3.5 Use `dict.get()` for Uncertain Keys

An occasionally forgotten convenience of the `dict` object is its `.get()` method. It's really handy, and code that takes other approaches is usually slightly cumbersome and less friendly.

Remember the students used in the prior section about sorting? Let's return to them. But let's add a few more of them:

```
students = [
    dict(name="Xian", grade="A-", age=10),
    dict(name="Jane", grade="B", age=12),
    dict(name="John", grade="C", age=15),
    dict(name="Pema", age=14),
    dict(name="Thandiwe", grade="B+")
```

We'd like to create a little report from our student list. A somewhat awkward approach to the missing data might be the following.

Look-before-you-leap approach (LBYL)

```
>>> print("| Name      | Grade | Age")
... print("+-----------+-------+-----")
... for student in students:
...     name = student['name'] if 'name' in student else "MISSING"
...     grade = student['grade'] if 'grade' in student else "PASS"
...     age = student['age'] if 'age' in student else "?"
...     print(f"| {name:9s} | {grade:<4s}  | {age}")
...
| Name      | Grade | Age
+-----------+-------+-----
| Xian      | A-    | 10
| Jane      | B     | 12
| John      | C     | 15
| Pema      | PASS  | 14
| Thandiwe  | B+    | ?
```

Despite what I warn in Chapter 4, *Advanced Python Usage*, there are times when forgiveness-not-permission makes code worse.

Easier-to-ask-forgiveness-than-permission approach (EAFP)

```
>>> print("| Name      | Grade | Age")
... print("+-----------+-------+-----")
... for student in students:
...     try:
...         name = student['name']
...     except KeyError:
...         name = "MISSING"
...     try:
...         grade = student['grade']
...     except KeyError:
...         grade = "PASS"
...     try:
...         age = student['age']
...     except KeyError:
...         age = "?"
...     print(f"| {name:9s} | {grade:<4s}  | {age}")
...
```