

Useful Modules

Scientific Programming with Python

Jonas Eschle & Nicola Chiapolini & Christian Elsasser

University of Zurich
Faculty of Science

16 July 2026



This work is licensed under the [Creative Commons Attribution-ShareAlike 4.0 License](https://creativecommons.org/licenses/by-sa/4.0/).

Strings and Files

difflib

- find differences between two sequences or strings

```
import difflib
```

```
start = ("bacon", "egg", "ham", "cheese")  
end = ["bacon", "egg", "spam", "cheese"]  
diff = difflib.ndiff(start, end)  
for line in diff:  
    print(line)
```

```
bacon  
egg  
- ham  
+ spam  
cheese
```

- calculate how close two sequences or strings are

```
options=["hannah", "anna", "hanna", "andrea", "maria"]  
close = difflib.get_close_matches("hana", options)  
print(close)
```

```
['hanna', 'hannah', 'anna']  
[(2, 3, 4), (1, 2, 3)]
```

```
options=[(1,2,3),(2,3,4),(3,4,5),(3,2,1)]  
close = difflib.get_close_matches([2,3], options)  
print(close)
```

difflib

- find differences between two sequences or strings

```
import difflib
```

```
start = ("bacon", "egg", "ham", "cheese")  
end = ["bacon", "egg", "spam", "cheese"]  
diff = difflib.ndiff(start, end)  
for line in diff:  
    print(line)
```

```
bacon  
egg  
- ham  
+ spam  
cheese
```

- calculate how close two sequences or strings are

```
options=["hannah", "anna", "hanna", "andrea", "maria"]  
close = difflib.get_close_matches("hana", options)  
print(close)
```

```
['hanna', 'hannah', 'anna']  
[(2, 3, 4), (1, 2, 3)]
```

```
options=[(1,2,3), (2,3,4), (3,4,5), (3,2,1)]  
close = difflib.get_close_matches([2,3], options)  
print(close)
```

pathlib

► find files on the file system

```
from pathlib import Path
```

```
tree = Path("tree")
matches = sorted(tree.glob("*"))
print(*matches, sep="\n")

print("")
matches = sorted(tree.rglob("*.py"))
print(*matches, sep="\n")
```

```
tree/fileA
tree/fileB.py
tree/subdir
```

```
tree/fileB.py
tree/subdir/file1.py
```

► object-oriented paths: build with /, read & write directly

```
f = Path("tree") / "subdir" / "file1.py"
print(f)
print(f.read_text())
```

```
tree/subdir/file1.py
print("I am file1")
```

► walk directory trees with Path.walk() (Python 3.12+)

pathlib

► find files on the file system

```
from pathlib import Path
```

```
tree = Path("tree")
matches = sorted(tree.glob("*"))
print(*matches, sep="\n")

print("")
matches = sorted(tree.rglob("*.py"))
print(*matches, sep="\n")
```

```
tree/fileA
tree/fileB.py
tree/subdir

tree/fileB.py
tree/subdir/file1.py
```

► object-oriented paths: build with /, read & write directly

```
f = Path("tree") / "subdir" / "file1.py"
print(f)
print(f.read_text())
```

```
tree/subdir/file1.py
print("I am file1")
```

► walk directory trees with Path.walk() (Python 3.12+)

pathlib

► find files on the file system

```
from pathlib import Path
```

```
tree = Path("tree")
matches = sorted(tree.glob("*"))
print(*matches, sep="\n")

print("")
matches = sorted(tree.rglob("*.py"))
print(*matches, sep="\n")
```

```
tree/fileA
tree/fileB.py
tree/subdir

tree/fileB.py
tree/subdir/file1.py
```

► object-oriented paths: build with /, read & write directly

```
f = Path("tree") / "subdir" / "file1.py"
print(f)
print(f.read_text())
```

```
tree/subdir/file1.py
print("I am file1")
```

► walk directory trees with Path.walk() (Python 3.12+)

pathlib (cont.)

- ▶ a path knows its parts (`.parent`, `.name`, `.stem`, `.suffix`) and can query the file system (`.exists()`, `.stat()`)

```
f = Path("tree") / "subdir" / "file1.py"
print(f.parent, f.name)
print(f.stem, f.suffix)
print(f.exists(), f.stat().st_size)
```

tree/subdir file1.py
file1 .py
True 20

- ▶ create, rename, delete: `mkdir`, `touch`, `rename`, `unlink` — with safety options

```
d = Path("sandbox")
d.mkdir(parents=True, exist_ok=True)
f = d / "a.txt"
f.touch()
f = f.rename(d / "b.txt")
print(sorted(d.iterdir()))
f.unlink(missing_ok=True)
d.rmdir()
print(d.exists())
```

[PosixPath('sandbox/b.txt')]
False

pathlib (cont.)

- ▶ a path knows its parts (`.parent`, `.name`, `.stem`, `.suffix`) and can query the file system (`.exists()`, `.stat()`)

```
f = Path("tree") / "subdir" / "file1.py"
print(f.parent, f.name)
print(f.stem, f.suffix)
print(f.exists(), f.stat().st_size)
```

```
tree/subdir file1.py
file1 .py
True 20
```

- ▶ create, rename, delete: `mkdir`, `touch`, `rename`, `unlink` — with safety options

```
d = Path("sandbox")
d.mkdir(parents=True, exist_ok=True)
f = d / "a.txt"
f.touch()
f = f.rename(d / "b.txt")
print(sorted(d.iterdir()))
f.unlink(missing_ok=True)
d.rmdir()
print(d.exists())
```

```
[PosixPath('sandbox/b.txt')]
False
```

tempfile

- ▶ scratch files and directories with automatic cleanup — unique names, nothing left behind

```
import tempfile
from pathlib import Path
```

```
with tempfile.TemporaryDirectory() as td:
    f = Path(td) / "scratch.txt"
    f.write_text("temporary data")
    print(f.read_text())
    print(f.exists())
print(f.exists())
```

```
temporary data
True
False
```

- ▶ also: `NamedTemporaryFile`, `gettempdir()`

tempfile

- ▶ scratch files and directories with automatic cleanup — unique names, nothing left behind

```
import tempfile
from pathlib import Path
```

```
with tempfile.TemporaryDirectory() as td:
    f = Path(td) / "scratch.txt"
    f.write_text("temporary data")
    print(f.read_text())
    print(f.exists())
print(f.exists())
```

```
temporary data
True
False
```

- ▶ also: NamedTemporaryFile, gettempdir()

Data Types

datetime

► proper representation of dates and times

```
import datetime as dt
```

```
t1 = dt.datetime(2017,9,11,13,15)  
print(t1)
```

2017-09-11 13:15:00

► reading and formatting of strings

```
t2 = dt.datetime.strptime("2017-09-11", "%Y-%m-%d")  
print(t2)
```

2017-09-11 00:00:00

Wed, 15 July 2026 19:23

```
t3 = dt.datetime.now()  
print(t3.strftime("%a, %d %B %Y %H:%M"))
```

datetime

► proper representation of dates and times

```
import datetime as dt
```

```
t1 = dt.datetime(2017,9,11,13,15)  
print(t1)
```

```
2017-09-11 13:15:00
```

► reading and formatting of strings

```
t2 = dt.datetime.strptime("2017-09-11", "%Y-%m-%d")  
print(t2)
```

```
2017-09-11 00:00:00
```

```
Wed, 15 July 2026 19:23
```

```
t3 = dt.datetime.now()  
print(t3.strftime("%a, %d %B %Y %H:%M"))
```

datetime (cont.)

► including timezones

```
t1 = dt.datetime(2017,9,11,13,15)
```

```
tz_cest = dt.timezone(dt.timedelta(hours=+2))  
t2 = dt.datetime(2017,9,11,13,15,tzinfo=tz_cest)  
print(t2)
```

```
2017-09-11 13:15:00+02:00  
no tzinfo: 13  
with tzinfo: 11
```

```
# Hour in UTC  
print("no tzinfo: ", t1.utctimetuple().tm_hour)  
print("with tzinfo:", t2.utctimetuple().tm_hour)
```

► named (IANA) timezones: zoneinfo (in the standard library)

```
from zoneinfo import ZoneInfo
```

```
zurich_dt = dt.datetime(2018, 3, 11, 19, 0,  
    tzinfo=ZoneInfo("Europe/Zurich"))  
print(zurich_dt)  
sydney_dt = zurich_dt.astimezone(  
    ZoneInfo("Australia/Sydney"))  
print(sydney_dt)
```

```
2018-03-11 19:00:00+01:00  
2018-03-12 05:00:00+11:00  
True
```

```
print(zurich_dt == sydney_dt)
```

datetime (cont.)

▶ including timezones

```
t1 = dt.datetime(2017,9,11,13,15)
```

```
tz_cest = dt.timezone(dt.timedelta(hours=+2))  
t2 = dt.datetime(2017,9,11,13,15,tzinfo=tz_cest)  
print(t2)
```

```
2017-09-11 13:15:00+02:00  
no tzinfo: 13  
with tzinfo: 11
```

```
# Hour in UTC  
print("no tzinfo: ", t1.utctimetuple().tm_hour)  
print("with tzinfo:", t2.utctimetuple().tm_hour)
```

▶ named (IANA) timezones: zoneinfo (in the standard library)

```
from zoneinfo import ZoneInfo
```

```
zurich_dt = dt.datetime(2018, 3, 11, 19, 0,  
    tzinfo=ZoneInfo("Europe/Zurich"))  
print(zurich_dt)  
sydney_dt = zurich_dt.astimezone(  
    ZoneInfo("Australia/Sydney"))  
print(sydney_dt)
```

```
2018-03-11 19:00:00+01:00  
2018-03-12 05:00:00+11:00  
True
```

```
print(zurich_dt == sydney_dt)
```

datetime (cont.)

- ▶ `timedelta` allows to do calculations

```
feb24 = dt.date(2024, 2, 28)
feb25 = dt.date(2025, 2, 28)
tdelta = dt.timedelta(days=1)
print(feb24+tdelta)
print(feb25+tdelta)
```

2024-02-29
2025-03-01

- ▶ a lot of useful functions: e.g. random date in interval

```
import random

start = dt.date(1990, 1,1).toordinal()
end = dt.date(2000, 1,1).toordinal()
rand = dt.date.fromordinal(random.randint(start, end))
print(rand)
```

1993-01-24

- ▶ integrates with dataframe libraries: `pandas` (`pd.Timestamp` is a `datetime` subclass) and `polars`

datetime (cont.)

- ▶ `timedelta` allows to do calculations

```
feb24 = dt.date(2024, 2, 28)
feb25 = dt.date(2025, 2, 28)
tdelta = dt.timedelta(days=1)
print(feb24+tdelta)
print(feb25+tdelta)
```

2024-02-29
2025-03-01

- ▶ a lot of useful functions: e.g. random date in interval

```
import random

start = dt.date(1990, 1, 1).toordinal()
end = dt.date(2000, 1, 1).toordinal()
rand = dt.date.fromordinal(random.randint(start, end))
print(rand)
```

1993-01-24

- ▶ integrates with dataframe libraries: `pandas` (`pd.Timestamp` is a `datetime` subclass) and `polars`

datetime (cont.)

- ▶ `timedelta` allows to do calculations

```
feb24 = dt.date(2024, 2, 28)
feb25 = dt.date(2025, 2, 28)
tdelta = dt.timedelta(days=1)
print(feb24+tdelta)
print(feb25+tdelta)
```

2024-02-29
2025-03-01

- ▶ a lot of useful functions: e.g. random date in interval

```
import random

start = dt.date(1990, 1, 1).toordinal()
end = dt.date(2000, 1, 1).toordinal()
rand = dt.date.fromordinal(random.randint(start, end))
print(rand)
```

1993-01-24

- ▶ integrates with dataframe libraries: `pandas` (`pd.Timestamp` is a `datetime` subclass) and `polars`

enum

- ▶ “magic strings” invite typos: "north" vs "North" — nothing complains
- ▶ an Enum is a named, fixed set of constants — invalid values fail immediately
- ▶ a StrEnum: members behave like strings

```
from enum import StrEnum
```

```
class Compass(StrEnum):
```

```
    n = "north"  
    ne = "northeast"  
    e = "east"  
    se = "southeast"  
    s = "south"  
    sw = "southwest"  
    w = "west"  
    nw = "northwest"
```

```
print(Compass.sw)  
print(Compass("north") == Compass.e)  
#print(Compass("osten")) # ValueError
```

```
southwest  
False
```

enum

- ▶ “magic strings” invite typos: "north" vs "North" — nothing complains
- ▶ an Enum is a named, fixed set of constants — invalid values fail immediately
- ▶ a StrEnum: members behave like strings

```
from enum import StrEnum
```

```
class Compass(StrEnum):
```

```
    n = "north"  
    ne = "northeast"  
    e = "east"  
    se = "southeast"  
    s = "south"  
    sw = "southwest"  
    w = "west"  
    nw = "northwest"
```

```
print(Compass.sw)
```

```
print(Compass("north") == Compass.e)
```

```
#print(Compass("osten")) # ValueError
```

```
southwest  
False
```

dataclasses

- ▶ classes for structured data without boilerplate: `__init__`, `__repr__` and `__eq__` are generated for you
- ▶ still a normal class — add methods as usual

```
from dataclasses import dataclass
```

```
@dataclass
```

```
class Article:
```

```
    title: str
```

```
    year: int
```

```
    def cite(self):
```

```
        return f"{self.title} ({self.year})"
```

```
dna = Article("DNA Structure", 1953)
```

```
print(dna)
```

```
print(dna.cite())
```

```
Article(title='DNA Structure', year=1953)
```

```
DNA Structure (1953)
```

dataclasses

- ▶ classes for structured data without boilerplate: `__init__`, `__repr__` and `__eq__` are generated for you
- ▶ still a normal class — add methods as usual

```
from dataclasses import dataclass
```

```
@dataclass
```

```
class Article:
```

```
    title: str
```

```
    year: int
```

```
    def cite(self):
```

```
        return f"{self.title} ({self.year})"
```

```
dna = Article("DNA Structure", 1953)
```

```
print(dna)
```

```
print(dna.cite())
```

```
Article(title='DNA Structure', year=1953)
```

```
DNA Structure (1953)
```

collections

► count occurrences

```
from collections import Counter
```

```
counts = Counter("abracadabra")  
print(counts)  
print(counts.most_common(2))
```

```
Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})  
[('a', 5), ('b', 2)]
```

► dictionaries with a default for missing keys

```
groups = defaultdict(list)  
for word in ["bacon", "egg", "beans", "ham"]:  
    groups[word[0]].append(word)  
print(dict(groups))
```

```
{'b': ['bacon', 'beans'], 'e': ['egg'], 'h': ['ham']}
```

► also: ChainMap, namedtuple (for structured data prefer dataclasses, see above)

collections

► count occurrences

```
from collections import Counter
```

```
counts = Counter("abracadabra")  
print(counts)  
print(counts.most_common(2))
```

```
Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})  
[('a', 5), ('b', 2)]
```

► dictionaries with a default for missing keys

```
groups = defaultdict(list)  
for word in ["bacon", "egg", "beans", "ham"]:  
    groups[word[0]].append(word)  
print(dict(groups))
```

```
{'b': ['bacon', 'beans'], 'e': ['egg'], 'h': ['ham']}
```

► also: ChainMap, namedtuple (for structured data prefer dataclasses, see above)

collections

► count occurrences

```
from collections import Counter
```

```
counts = Counter("abracadabra")  
print(counts)  
print(counts.most_common(2))
```

```
Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})  
[('a', 5), ('b', 2)]
```

► dictionaries with a default for missing keys

```
groups = defaultdict(list)  
for word in ["bacon", "egg", "beans", "ham"]:  
    groups[word[0]].append(word)  
print(dict(groups))
```

```
{'b': ['bacon', 'beans'], 'e': ['egg'], 'h': ['ham']}
```

► also: ChainMap, namedtuple (for structured data prefer dataclasses, see above)

collections (cont.)

- ▶ deque: double-ended queue, fast appends and pops on both ends

```
from collections import deque
```

```
d = deque(["b", "c"])
d.appendleft("a")
d.append("d")
print(d)
print(d.popleft(), d.pop())
print(d)
```

```
deque(['a', 'b', 'c', 'd'])
a d
deque(['b', 'c'])
```

- ▶ maxlen=N keeps only the last N items (ring buffer)

collections (cont.)

- ▶ deque: double-ended queue, fast appends and pops on both ends

```
from collections import deque
```

```
d = deque(["b", "c"])
d.appendleft("a")
d.append("d")
print(d)
print(d.popleft(), d.pop())
print(d)
```

```
deque(['a', 'b', 'c', 'd'])
a d
deque(['b', 'c'])
```

- ▶ maxlen=N keeps only the last N items (ring buffer)

pickle

- save (almost) any Python object to a file and load it back — files must be opened in binary mode (wb/rb)

```
import pickle

data = {"run": 7, "values": [1.5, 2.5]}
with open("data.pkl", "wb") as f:
    pickle.dump(data, f)
with open("data.pkl", "rb") as f:
    loaded = pickle.load(f)
print(loaded)
print(loaded == data)
```

{'run': 7, 'values': [1.5, 2.5]}

True

- never unpickle untrusted data — loading can execute arbitrary code

pickle

- save (almost) any Python object to a file and load it back — files must be opened in binary mode (wb/rb)

```
import pickle
```

```
data = {"run": 7, "values": [1.5, 2.5]}
```

```
with open("data.pkl", "wb") as f:
```

```
    pickle.dump(data, f)
```

```
with open("data.pkl", "rb") as f:
```

```
    loaded = pickle.load(f)
```

```
print(loaded)
```

```
print(loaded == data)
```

```
{'run': 7, 'values': [1.5, 2.5]}
```

```
True
```

- **never unpickle untrusted data** — loading can execute arbitrary code

Functional Tools

itertools

- ▶ lots of tools to work with sequences

- ▶ chain sequences

```
from itertools import chain
```

```
res = [v for v in chain('ABC', 'DEF')]  
print(res)
```

```
['A', 'B', 'C', 'D', 'E', 'F']
```

- ▶ infinitely cycle sequences

```
gen = cycle('ABC')  
print([next(gen) for _ in range(5)])
```

```
['A', 'B', 'C', 'A', 'B']
```

- ▶ generate all permutations

```
res = ["".join(v) for v in permutations('ABC')]  
print(res)
```

```
['ABC', 'ACB', 'BAC', 'BCA', 'CAB', 'CBA']
```

itertools

- ▶ lots of tools to work with sequences

- ▶ chain sequences

```
from itertools import chain
```

```
res = [v for v in chain('ABC', 'DEF')]  
print(res)
```

```
['A', 'B', 'C', 'D', 'E', 'F']
```

- ▶ infinitely cycle sequences

```
gen = cycle('ABC')  
print([next(gen) for _ in range(5)])
```

```
['A', 'B', 'C', 'A', 'B']
```

- ▶ generate all permutations

```
res = ["".join(v) for v in permutations('ABC')]  
print(res)
```

```
['ABC', 'ACB', 'BAC', 'BCA', 'CAB', 'CBA']
```

itertools

- ▶ lots of tools to work with sequences

- ▶ chain sequences

```
from itertools import chain
```

```
res = [v for v in chain('ABC', 'DEF')]  
print(res)
```

```
['A', 'B', 'C', 'D', 'E', 'F']
```

- ▶ infinitely cycle sequences

```
gen = cycle('ABC')  
print([next(gen) for _ in range(5)])
```

```
['A', 'B', 'C', 'A', 'B']
```

- ▶ generate all permutations

```
res = ["".join(v) for v in permutations('ABC')]  
print(res)
```

```
['ABC', 'ACB', 'BAC', 'BCA', 'CAB', 'CBA']
```

functools

► cache results of expensive functions

```
from functools import cache
```

```
@cache
```

```
def fib(n):
```

```
    if n < 2:
```

```
        return n
```

```
    return fib(n - 1) + fib(n - 2)
```

```
print(fib(80))
```

23416728348467685

► fix arguments of a function

```
int2 = partial(int, base=2)
```

```
print(int2("1011"))
```

```
print([int2(s) for s in ["10", "111"]])
```

11

[2, 7]

► writing your own decorators? use @functools.wraps

functools

► cache results of expensive functions

```
from functools import cache
```

```
@cache
```

```
def fib(n):
```

```
    if n < 2:
```

```
        return n
```

```
    return fib(n - 1) + fib(n - 2)
```

```
print(fib(80))
```

23416728348467685

► fix arguments of a function

```
int2 = partial(int, base=2)
```

```
print(int2("1011"))
```

```
print([int2(s) for s in ["10", "111"]])
```

11

[2, 7]

► writing your own decorators? use `@functools.wraps`

functools

► cache results of expensive functions

```
from functools import cache
```

```
@cache
```

```
def fib(n):
```

```
    if n < 2:
```

```
        return n
```

```
    return fib(n - 1) + fib(n - 2)
```

```
print(fib(80))
```

23416728348467685

► fix arguments of a function

```
int2 = partial(int, base=2)
```

```
print(int2("1011"))
```

```
print([int2(s) for s in ["10", "111"]])
```

11

[2, 7]

► writing your own decorators? use @functools.wraps

Settings and User Input

argparse

- Parse command-line arguments to adjust program execution easily

```
import argparse

parser = argparse.ArgumentParser(description="Process some integers.")
parser.add_argument("integers", metavar="N", type=int, nargs="+",
                    help="an integer for the accumulator")
parser.add_argument("--sum", dest="accumulate", action="store_const",
                    const=sum, default=max,
                    help="sum the integers (default: find the max)")
args = parser.parse_args()

print(args.accumulate(args.integers))
```

```
$ python argparsedemo.py 1 2 3 4
```

```
4
```

```
$ python argparsedemo.py --sum 1 2 3 4
```

```
10
```

click

- Makes writing CLIs (Command Line Interfaces) even easier.

```
import click

@click.command()
@click.argument("integers", type=int, nargs=-1, required=True)
@click.option("--sum/--max", "-s/-m", "sum_", default=False,
              help="use sum or max (default: max)")
def process(sum_, integers):
    accumulator = sum if sum_ else max
    print(accumulator(integers))

process()
```

```
$ python clickdemo.py --sum 1 2 3 4
```

```
10
```

configparser

► Load configuration from files.

```
import configparser

conf = configparser.ConfigParser()
conf.read(["config.ini"])

accumulator = max
if conf["setup"].getboolean("sum", False):
    accumulator = sum

integers = [
    int(v) for v in (conf["values"]["ints"]).split(",")
]

print(accumulator(integers))
```

```
[setup]
sum = True

[values]
ints = 1,2,3
```

tomllib

- ▶ load configuration from TOML files
(the modern config format, e.g. `pyproject.toml`)

```
import tomllib

with open("config.toml", "rb") as f:
    conf = tomllib.load(f)

accumulator = max
if conf["setup"]["sum"]:
    accumulator = sum

print(accumulator(conf["values"]["ints"]))
```

```
[setup]
sum = false

[values]
ints = [1, 22, 3]
```

- ▶ values are typed: no manual parsing of numbers, lists, booleans
- ▶ in the standard library since Python 3.11 (reading only)

Output

logging

- ▶ print-debugging you can switch on and off: levels (DEBUG ... CRITICAL) select the verbosity in one place — keep debug statements in the code
- ▶ messages carry timestamp and origin, and can go to the console and to files
- ▶ control output verbosity

```
import logging

logging.basicConfig(
    format='%(levelname)s - %(name)s - %(message)s',
    level=logging.WARNING # this is the default
)
```

```
# recommended setup, works well with imports
logger = logging.getLogger(__name__)
```

```
logger.debug("This output is only for debugging")
logger.error("There was an error.")
```

```
ERROR - __main__ - There was an error.
```

logging

- ▶ print-debugging you can switch on and off: levels (DEBUG ... CRITICAL) select the verbosity in one place — keep debug statements in the code
- ▶ messages carry timestamp and origin, and can go to the console and to files
- ▶ control output verbosity

```
import logging

logging.basicConfig(
    format='%(levelname)s - %(name)s - %(message)s',
    level=logging.WARNING # this is the default
)
```

```
# recommended setup, works well with imports
logger = logging.getLogger(__name__)
```

```
logger.debug("This output is only for debugging")
logger.error("There was an error.")
```

ERROR - __main__ - There was an error.

logging (cont.)

► detailed control of output

```
log = logging.getLogger(__name__)
# set lowest log-level, handlers can only
# increase threshold/reduce verbosity
log.setLevel(logging.DEBUG)

# add an output-channel for stderr
log.addHandler(logging.StreamHandler())
# reduce output for last handler
log.handlers[-1].setLevel(logging.ERROR)

# create and configure output to a file
# (use "a" instead of "w" to append)
logfile = logging.FileHandler("demo.log", "w")
# define format of Log-lines
# - see `pydoc3 logging.Formatter` for variables
# - extra variables (e.g. `%(var)s`) must be passed
#   via keyword argument `extra`
logfile.setFormatter(logging.Formatter(
    "%(asctime)s -- %(levelname)s, %(var)s: %(message)s"
))
log.addHandler(logfile)
```

```
log.error("This is bad", extra={"var": 42})
log.info("Info: %s", "busy", extra={"var": 0})
```

console:

This is bad

logfile

```
2026-07-15 19:23:21,053 -- ERROR, 42: This is bad
2026-07-15 19:23:21,053 -- INFO, 0: Info: busy
```

textwrap

► wrapping and filling text

```
import textwrap
```

```
text = """This is an example  
text that should be wrapped  
into longer lines, ignoring any  
newlines in the original.  
"""
```

```
res = textwrap.fill(text, width=30)  
print(res)
```

```
print()
```

```
res = textwrap.indent(res, prefix=' # ')  
print(res)
```

This is an example text that
should be wrapped into
longer lines, ignoring any
newlines in the original.

```
# This is an example text that  
# should be wrapped into  
# longer lines, ignoring any  
# newlines in the original.
```

texttable

► output tables to terminal

```
from texttable import Texttable

output = Texttable(max_width=30)
output.set_cols_align(["l", "r", "r"])
output.add_row(["n", "2*n", "n**2"])
for n in range(5):
    output.add_row([n, 2*n, n**2])
print(output.draw())
```

```
+---+-----+-----+
| n | 2*n | n**2 |
+---+-----+-----+
| 0 |  0 |   0 |
+---+-----+-----+
| 1 |  2 |   1 |
+---+-----+-----+
| 2 |  4 |   4 |
+---+-----+-----+
| 3 |  6 |   9 |
+---+-----+-----+
| 4 |  8 |  16 |
+---+-----+-----+
```

► more powerful alternatives: tabulate and rich

Interacting with the Web

requests

- ▶ interact with things reachable over the internet
- ▶ HTTP get requests: e.g. fetching a website

```
import requests
```

```
url = "https://www.uzh.ch"  
response = requests.get(url)  
response.raise_for_status()
```

```
lines = response.text.split("\n")  
lines = [l for l in lines if l.strip() != ""]  
for line in lines[:5]:  
    print(line)
```

- ▶ HTTP post requests: e.g. filling a form

```
res = requests.post(  
    "https://httpbin.org/post",  
    data={"user": "me", "pass": "secret"}  
)  
print("Status:", res.status_code)
```

```
<!DOCTYPE html>  
<html lang="de" data-template="ct01">  
<head>  
    <meta charset="utf-8">  
    <meta name="viewport" content="width=device-width, >
```

Status: 200

requests

- ▶ interact with things reachable over the internet
- ▶ HTTP get requests: e.g. fetching a website

```
import requests
```

```
url = "https://www.uzh.ch"  
response = requests.get(url)  
response.raise_for_status()
```

```
lines = response.text.split("\n")  
lines = [l for l in lines if l.strip() != ""]  
for line in lines[:5]:  
    print(line)
```

- ▶ HTTP post requests: e.g. filling a form

```
res = requests.post(  
    "https://httpbin.org/post",  
    data={"user": "me", "pass": "secret"}  
)  
print("Status:", res.status_code)
```

```
<!DOCTYPE html>  
<html lang="de" data-template="ct01">  
<head>
```

```
    <meta charset="utf-8">
```

```
    <meta name="viewport" content="width=device-width, ...>
```

Status: 200

bs4 / lxml

- ▶ extract elements from XML/HTML documents: e.g. links

```
from bs4 import BeautifulSoup

content = BeautifulSoup(response.text, "lxml")
links = content.find_all(name="a", href=True) # by tag name
#links = content.select("a.Link") # by css selector
for link in links[:5]:
    print(link["href"])
```

```
#main-content
https://www.uzh.ch
/de/search.html
https://www.students.uzh.ch
https://www.staff.uzh.ch
```

- ▶ alternative option lxml

```
import lxml.html

content = lxml.html.fromstring(response.text)
links = content.findall(".*[a[@href]]") # by xpath
for link in links[:5]:
    print(link.get("href"))
```

```
#main-content
https://www.uzh.ch
/de/search.html
https://www.students.uzh.ch
https://www.staff.uzh.ch
```

selenium

- ▶ remote control a real browser (for JavaScript and other dynamic content)

```
from selenium import webdriver
from selenium.webdriver.common.by import By

# selenium manages the browser drivers itself
driver = webdriver.Chrome()
driver.get("https://www.uzh.ch")
links = driver.find_elements(By.TAG_NAME, "a")
links[-1].click()
```

Demo

Running External Commands

subprocess

► run external command

```
import subprocess
```

```
result = subprocess.run(["uname", "-om"])  
print("res:", result)
```

```
x86_64 GNU/Linux
```

```
res: CompletedProcess(args=['uname', '-om'], returncode=0)
```

► capture output to access it from python

```
result = subprocess.run(  
    ["uname", "-om"],  
    capture_output=True,  
    text=True  
)  
print("___")  
print(result.stdout)  
print("___")
```

```
---
```

```
x86_64 GNU/Linux
```

```
---
```

► check=True raises an error if the command fails

subprocess

► run external command

```
import subprocess
```

```
result = subprocess.run(["uname", "-om"])  
print("res:", result)
```

```
x86_64 GNU/Linux
```

```
res: CompletedProcess(args=['uname', '-om'], returncode=0)
```

► capture output to access it from python

```
result = subprocess.run(  
    ["uname", "-om"],  
    capture_output=True,  
    text=True  
)  
print("----")  
print(result.stdout)  
print("----")
```

```
---
```

```
x86_64 GNU/Linux
```

```
---
```

► check=True raises an error if the command fails

subprocess

► run external command

```
import subprocess
```

```
result = subprocess.run(["uname", "-om"])  
print("res:", result)
```

```
x86_64 GNU/Linux
```

```
res: CompletedProcess(args=['uname', '-om'], returncode=0)
```

► capture output to access it from python

```
result = subprocess.run(  
    ["uname", "-om"],  
    capture_output=True,  
    text=True  
)  
print("----")  
print(result.stdout)  
print("----")
```

```
---
```

```
x86_64 GNU/Linux
```

```
---
```

► check=True raises an error if the command fails

subprocess (cont.)

- ▶ metacharacters are not interpreted unless `shell=True`

```
ext = "py"
res = subprocess.run(
    [f"ls -l *.{ext}"], # single element
    shell=True # RISKY!
)
```

capture.py
run.py
shell.py
stdin.py

```
# using:
# ext = input("ext: ")
# creates a shell injection vulnerability
```

Demo

- ▶ pass data to standard input with `input=...`

```
data = """# A title
And a simple paragraph
with some text.
"""
res = subprocess.run(["pandoc", "-t", "html"],
    input=data.encode()
)
```

```
<h1 id="a-title">A title</h1>
<p>And a simple paragraph with some text.</p>
```

subprocess (cont.)

- ▶ metacharacters are not interpreted unless `shell=True`

```
ext = "py"
res = subprocess.run(
    [f"ls -l *.{ext}"], # single element
    shell=True # RISKY!
)
```

capture.py
run.py
shell.py
stdin.py

```
# using:
# ext = input("ext: ")
# creates a shell injection vulnerability
```

Demo

- ▶ pass data to standard input with `input=...`

```
data = """# A title
And a simple paragraph
with some text.
"""
res = subprocess.run(["pandoc", "-t", "html"],
    input=data.encode()
)
```

```
<h1 id="a-title">A title</h1>
<p>And a simple paragraph with some text.</p>
```

Running Code in Parallel

multiprocessing

- ▶ use all CPU cores by running multiple *processes*
- ▶ `Pool.map`: apply a function to many inputs in parallel

```
import multiprocessing as mp
```

```
def square(n):  
    return n * n
```

```
if __name__ == "__main__":  
    with mp.Pool(4) as pool:  
        res = pool.map(square, range(8))  
    print(res)
```

[0, 1, 4, 9, 16, 25, 36, 49]

multiprocessing

- ▶ use all CPU cores by running multiple *processes*
- ▶ `Pool.map`: apply a function to many inputs in parallel

```
import multiprocessing as mp

def square(n):
    return n * n

if __name__ == "__main__":
    with mp.Pool(4) as pool:
        res = pool.map(square, range(8))
    print(res)
```

[0, 1, 4, 9, 16, 25, 36, 49]

multiprocessing: parallelism is dangerous

- ▶ processes run in unpredictable order — unsynchronized read-modify-write on shared data loses updates (race condition)

```
import multiprocessing as mp

def unsafe(count, lock):
    for _ in range(100_000):
        count[0] += 1

def safe(count, lock):
    for _ in range(100_000):
        with lock:
            count[0] += 1
```

```
def run(fun):
    count = mp.Array("i", [0])
    lock = mp.Lock()
    ps = [mp.Process(target=fun,
                     args=(count, lock))
          for _ in range(2)]
    for p in ps:
        p.start()
    for p in ps:
        p.join()
    print(fun.__name__, count[0])
```

output:

```
unsafe 108056
safe 200000
```

- ▶ without the lock the result is wrong — and different on every run

multiprocessing: parallelism is dangerous

- ▶ processes run in unpredictable order — unsynchronized read-modify-write on shared data loses updates (race condition)

```
import multiprocessing as mp

def unsafe(count, lock):
    for _ in range(100_000):
        count[0] += 1

def safe(count, lock):
    for _ in range(100_000):
        with lock:
            count[0] += 1
```

```
def run(fun):
    count = mp.Array("i", [0])
    lock = mp.Lock()
    ps = [mp.Process(target=fun,
                     args=(count, lock))
          for _ in range(2)]
    for p in ps:
        p.start()
    for p in ps:
        p.join()
    print(fun.__name__, count[0])
```

output:

```
unsafe 108056
safe 200000
```

- ▶ without the lock the result is wrong — and different on every run

multiprocessing: rules of the game

- ▶ always guard with `if __name__ == "__main__":` — worker processes re-import your file
- ▶ arguments and results travel between processes by pickling — they must be picklable
- ▶ starting a process costs milliseconds — only worth it for real work
- ▶ also: `concurrent.futures.ProcessPoolExecutor`; threads for waiting on I/O

multiprocessing: rules of the game

- ▶ always guard with `if __name__ == "__main__":` — worker processes re-import your file
- ▶ arguments and results travel between processes by pickling — they must be picklable
- ▶ starting a process costs milliseconds — only worth it for real work
- ▶ also: `concurrent.futures.ProcessPoolExecutor`; threads for waiting on I/O

multiprocessing: rules of the game

- ▶ always guard with `if __name__ == "__main__":` — worker processes re-import your file
- ▶ arguments and results travel between processes by pickling — they must be picklable
- ▶ starting a process costs milliseconds — only worth it for real work
- ▶ also: `concurrent.futures.ProcessPoolExecutor`; threads for waiting on I/O

multiprocessing: rules of the game

- ▶ always guard with `if __name__ == "__main__":` — worker processes re-import your file
- ▶ arguments and results travel between processes by pickling — they must be picklable
- ▶ starting a process costs milliseconds — only worth it for real work
- ▶ also: `concurrent.futures.ProcessPoolExecutor`; threads for waiting on I/O