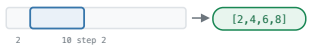
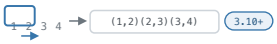
Python Standard Library Power Tools

A tour of the batteries already in Python, like itertools, collections, functools, and dataclasses, with nothing to pip install.

itertools: iterate without building lists

Lazy building blocks that stream items instead of materializing lists.

PURPOSE	COMMAND	RESULT
Concatenate iterables end to end.	`list(itertools.chain([1, 2], [3, 4]))`	
Group adjacent equal keys (sort first!).	`[(k, list(g)) for k, g in itertools.groupby(rows, key=...)]`	
Slice an iterator (no indexing).	`list(itertools.islice(range(100), 2, 10, 2))`	
Cartesian product (nested loops).	`list(itertools.product([1, 2], ["a", "b"]))`	
Running totals / scans.	`list(itertools.accumulate([1, 2, 3, 4]), operator.mul)`	
Consecutive pairs (sliding window).	`list(itertools.pairwise([1, 2, 3, 4]))`	

Lazy iterators stream on demand and stay an iterator until you wrap them in list(...); chain, islice, product, groupby, accumulate, pairwise

collections: tally, group, queue, record

Specialized containers that replace boilerplate dict and list code.

PURPOSE	COMMAND	RESULT
Count occurrences (a multiset).	`Counter("mississippi").most_common(2)`	
Auto-seed missing keys.	`dd = defaultdict(list); dd["x"].append(1)`	
Fast appends/pops at both ends.	`deque([1, 2, 3], maxlen=3).appendleft(0)`	
Rotate a ring buffer.	`dq.rotate(1)`	
Lightweight typed record (tuple).	`Point = namedtuple("Point", ["x","y"]); p = Point(1, 2)`	
Layer multiple dicts as one view.	`dict(ChainMap(overrides, defaults))`	

Reach for a purpose-built container instead of hand-rolling dict and list boilerplate: tally, group, queue, record, layer.

functools

Wrappers that memoize, pre-fill arguments, fold, and preserve metadata.

PURPOSE	COMMAND	RESULT
Memoize an expensive pure function.	`@lru_cache(maxsize=None)` `def fib(n): ...`	
Unbounded cache shorthand.	`@cache # = lru_cache(maxsize=None)`	
Pre-fill arguments.	`add10 = partial(operator.add, 10)`	
Fold a sequence to one value.	`reduce(operator.mul, [1, 2, 3, 4], 1)`	
Cache a computed attribute once.	`@cached_property` `def val(self): ...`	
Keep wrapper name/docstring.	`@wraps(fn) # inside a decorator`	

`@cache / @lru_cache memoize, partial pre-fills args, reduce folds, @cached_property computes once; always @wraps your own decorators`

datetime: instants, dates, and durations

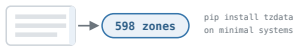
Always make datetimes timezone-aware; never use the deprecated utcnow().

PURPOSE	COMMAND	RESULT
Current instant, UTC-aware.	`dt.datetime.now(dt.UTC)`	
Today's calendar date.	`dt.date.today()`	
A duration to add or subtract.	`dt.timedelta(days=1, hours=2, minutes=30)`	
Difference between two instants.	`later - earlier`	
Parse an ISO-8601 string.	`dt.datetime.fromisoformat(` `"2026-06-18T09:30:00-05:00")`	
Format out / parse a custom layout.	`aware.strftime("%Y-%m-%d %H:%M %Z")` `dt.datetime.strptime(s, fmt)`	

`Naive vs aware: prefer aware. now(dt.UTC) over utcnow(); timedelta for spans; fromisoformat / strftime / strptime to round-trip`

zoneinfo: real IANA time zones

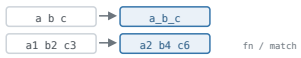
Attach and convert IANA zones with the stdlib (3.9+); no third-party pytz.

PURPOSE	COMMAND	RESULT
Load a named zone.	`ZoneInfo("America/Chicago")`	
Make a naive datetime aware.	`dt.datetime(2026, 6, 18, 9, 30, tzinfo=ZoneInfo("America/Chicago"))`	
Convert to another zone.	`aware.astimezone(ZoneInfo("…/New_York"))`	
Anchor everything in UTC, display local.	`dt.datetime.now(dt.UTC) .astimezone(ZoneInfo("Europe/Paris"))`	
DST handled automatically.	`ZoneInfo("America/Chicago")`	
List installed zones.	`len(zoneinfo.available_timezones())`	

ZoneInfo("Area/City") loads a real IANA zone with full DST rules; store and compute in UTC, convert to a local zone only for display

re: regular expressions

Compile once, then match, search, capture groups, and substitute.

PURPOSE	COMMAND	RESULT
Compile a reusable pattern.	`pat = re.compile(r"(?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2})")`	
Match at the start vs anywhere.	`pat.match(s)` `re.search(r"\d+", s)`	
Pull out captured groups.	`m.group("year"), m.groupdict()`	
Find every occurrence.	`re.findall(r"\d+", "a1 b22 c333")`	
Substitute (literal or callable).	`re.sub(r"\s+", "_", s)` `re.sub(r"\d+", lambda m: ..., s)`	
Split on a pattern.	`re.split(r"[,;]\s*", "a, b; c,d")`	

re.compile builds a reusable pattern; match anchors at the start, search scans anywhere, and sub replaces with a string or a callable

contextlib: clean resource management

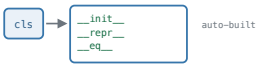
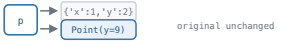
Build with-blocks, swallow expected errors, and stack cleanups.

PURPOSE	COMMAND	RESULT
Write a context manager from a generator.	`@contextmanager def tag(name): ...yield`	
Swallow an expected exception.	`with suppress(FileNotFoundError):`	
Capture printed output.	`with redirect_stdout(buf): print(...)`	
Manage a dynamic number of resources.	`with ExitStack() as s: s.enter_context(...)`	
Guarantee close on any object.	`with closing(thing) as t: ...`	
Reuse one manager as a decorator.	`class T(contextlib.ContextDecorator):`	

@contextmanager builds a with-block from one yield; suppress drops expected errors; ExitStack unwinds resources in reverse (LIFO)

dataclasses: typed records without boilerplate

Annotate fields; get __init__, __repr__, and __eq__ generated for you.

PURPOSE	COMMAND	RESULT
Generate a record from annotations.	`@dataclass class Point: x: int; y: int = 0`	
Mutable default the safe way.	`tags: list[str] = field(default_factory=list)`	
Immutable, memory-light record.	`@dataclass(frozen=True, slots=True)`	
Make instances sortable.	`@dataclass(order=True)`	
Convert to a plain dict (or copy).	`asdict(p) and replace(p, y=9)`	
Derive a field after init.	`area: float = field(init=False)` `# __post_init__`	

@dataclass turns annotated fields into a typed record; field() handles defaults, frozen/slots/order tune behavior, __post_init__ derives