TypeScript for Python devs/Python for TS devs

Javier Candeira for Melb Py Meetup - 11 June 2026

TS Arrays (Python `list`)

Python	TypeScript
`lst.append(x)`	`arr.push(x)`
`lst.pop()`	`arr.pop()`
`lst.remove(x)`	`arr = arr.filter(i ⇒ i ≠= x)`
`x in lst`	`arr.includes(x)`
`len(lst)`	`arr.length`
`lst[1:3]`	`arr.slice(1, 3)`
`[f(x) for x in lst]`	`arr.map(x ⇒ f(x))`
`[x for x in lst if p(x)]`	`arr.filter(x ⇒ p(x))`
`any(p(x) for x in lst)`	`arr.some(x ⇒ p(x))`
`all(p(x) for x in lst)`	`arr.every(x ⇒ p(x))`
`sum(lst)`	`arr.reduce((a, b) ⇒ a + b, 0)`

TS Dicts (Python `dict`)

Use `Map` when keys are dynamic. Use plain objects `{}` for fixed/known keys.

Python	TypeScript (`Map`)
`d = {}`	`const m = new Map()`
`d[k] = v`	`m.set(k, v)`
`d[k]`	`m.get(k)` — returns `undefined` if missing
`k in d`	`m.has(k)`
`del d[k]`	`m.delete(k)`
`d.keys()`	`m.keys()`
`d.values()`	`m.values()`
`d.items()`	`m.entries()`
`d.get(k, default)`	`m.get(k) ?? default`

TS Sets (Python `set`)

Python	TypeScript
`s = {1, 2}`	`const s = new Set([1, 2])`
`s.add(x)`	`s.add(x)`
`s.remove(x)`	`s.delete(x)`
`x in s`	`s.has(x)`
`len(s)`	`s.size`

TS does Python defaultdict(list|dict|set) as pattern

```ts
// Python: d = defaultdict(list); d[k].append(x)
if (!m.has(k)) m.set(k, []);
m.get(k)!.push(x);  // type-checker needs the non-null assertion

// Writing it as the germ of a Python-style defaultdict:
function getOrSet<K, V>(
  map: Map<K, V>,
  key: K,
  // new () ⇒ V means "a callable constructor
  // that takes no arguments and returns a V"
  // This is the type of a Python class-as-factory
  defaultConstructor: new () ⇒ V) ⇒ {
    if (!map.has(key)) map.set(key, new V);
    return map.get(key)!;
}
// defaultVal third argument is an array: defaultdict(list)
getOrSet(map, key, Array).push(x);
// defaultVal third argument is a Map: defaultdict(dict)
getOrSet(map, key, Map).set(innerKey, x);
// defaultVal third argument is a Set: defaultdict(set)
getOrSet(map, key, Set).add(x);
```

Common Operations

Sorting with a key

```ts
# Python
lst.sort(key=lambda x: x.name)
```

```ts
// TypeScript — comparator, not key function
arr.sort((a, b) ⇒ a.name.localeCompare(b.name));  // strings
arr.sort((a, b) ⇒ a.age - b.age);                  // numbers
arr.sort((a, b) ⇒ b.age - a.age);                  // reverse
```

Comparator returns: negative → a first, positive → b first, zero → equal.

Branded Types

Make two values of the same primitive type nominally distinct — the compiler rejects mixing them without an explicit cast.

```python
# Python — NewType (checked by mypy/pyright, not at runtime)
from typing import NewType

W   = NewType("W", float)
kW  = NewType("kW", float)
Wh  = NewType("Wh", float)
kWh = NewType("kWh", float)

reading: kWh = kWh(12.5)
# passing W where kWh expected → mypy error
```

```typescript
// TypeScript — phantom brand via intersection
type W   = number & { readonly _brand: "W" };
type kW  = number & { readonly _brand: "kW" };
type Wh  = number & { readonly _brand: "Wh" };
type kWh = number & { readonly _brand: "kWh" };

// In test data / call sites — cast inline, no helper needed:
const reading = 12.5 as kWh;
// passing W where kWh expected → tsc error
```

Conversion helpers (optional):

```typescript
const wToKw   = (w: W): kW   => (w / 1000) as kW;
const whToKwh = (wh: Wh): kWh => (wh / 1000) as kWh;
```

Copy vs Deep Copy

```typescript
// Shallow copy — like a[:]  or copy.copy()
const b = [...a];            // array
const o = { ...obj };        // object
const m = new Map(original) // Map

// These are NOT copies — same reference:
const b = a;
const o = obj;
```

For deep copy (like `copy.deepcopy()`), TS has no built-in. Common options:

```typescript
// Works for plain JSON-serialisable data (no functions, no Map/Set)
const deep = JSON.parse(JSON.stringify(obj));
```

```ts
// Modern environments (Node 17+, browsers)
const deep = structuredClone(obj);  // handles Map, Set, Date too
```

`structuredClone` is the right answer in 2024+. Avoid the `JSON` round-trip unless you know your data is plain.

Async

Coroutines vs Promises

Calling an async function without `await` produces different things in each language:

Coroutines vs Promises	Python	TypeScript
Name	coroutine	Promise
Starts running immediately?	**No** — lazy, does nothing until awaited	**Yes** — eager, starts on creation
Reusable?	No — one-shot	No — one-shot
Can be awaited multiple times?	No	Yes — resolves to same value each time
Unhandled warning?	Yes (`RuntimeWarning`)	Yes (`unhandledRejection`)

```python
coro = fetch()        # nothing happens yet
await coro            # now it runs
```

```ts
const p = fetch();    // already running
await p;              // wait for it to finish
```

Because Promises are eager, in TS you can start multiple operations before awaiting any of them and they run concurrently:

```ts
const p1 = fetch("a");   // started
const p2 = fetch("b");   // started concurrently
await p1;
await p2;
```

In plain Python, two bare `await` calls in sequence run sequentially — you need asyncio primitives to get concurrency (see next section).

Asyncio vs TS

Asyncio adds scheduling, concurrency primitives, and cancellation on top of bare coroutines. This table maps asyncio concepts to their TS equivalents.

Common Operations	Python asyncio	TypeScript
Schedule coroutine (don't await)	`asyncio.create_task(coro)`	`f()` — Promise is already running
Run event loop	`asyncio.run(main())`	runtime handles it (Node / browser)
Concurrent, call order	`asyncio.gather(f(), g())`	`Promise.all([f(), g()])`
Concurrent, completion order	`asyncio.as_completed(coros)`	no built-in — implement manually

Common Operations	Python asyncio	TypeScript
Bounded concurrency	`asyncio.Semaphore(n)`	no built-in — implement with a counter
Cancel a task	`task.cancel()`	`AbortController` + `signal.abort()`
Sleep	`await asyncio.sleep(1)`	`await new Promise(r ⇒ setTimeout(r, 1000))`
Timeout	`asyncio.wait_for(coro, timeout=5)`	`Promise.race([f(), AbortSignal.timeout(ms)])`
Mutex	`asyncio.Lock()`	no built-in — promise chain or library

Common async operations

Python column assumes asyncio throughout.

Operation	Python asyncio	TypeScript
Define async function	`async def f():`	`async function f()`
Await a single result	`await f()`	`await f()`
Concurrent, results in **call order**	`asyncio.gather(f(), g())`	`Promise.all([f(), g()])`
Concurrent, results in **completion order**	`asyncio.as_completed(coros)`	no built-in — implement manually
Concurrent, collect all (no throw)	`asyncio.gather( … , return_exceptions=True)`	`Promise.allSettled([f(), g()])`
First to complete (any result)	`asyncio.wait( … , return_when=FIRST_COMPLETED)`	`Promise.race([f(), g()])`
First to succeed (ignore errors)	n/a (manual)	`Promise.any([f(), g()])`
Bounded concurrency	`asyncio.Semaphore(n)` + `async with sem:`	no built-in — implement with a counter
Cancel remaining tasks	`task.cancel()` on `asyncio.Task`	`AbortController` + `signal.abort()`
Retry with backoff	manual loop + `await asyncio.sleep(delay)`	manual loop + `await sleep(delay)`
Fire and forget	`asyncio.create_task(f())`	`f()` (just don't await)
Sleep	`await asyncio.sleep(1)`	`await new Promise(r ⇒ setTimeout(r, 1000))`
Timeout	`asyncio.wait_for(f(), timeout=5)`	`Promise.race([f(), AbortSignal.timeout(ms)])`
Async generator (produce)	`async def g(): yield x`	`async function* g() { yield x }`
Async generator (consume)	`async for item in g():`	`for await (const item of g())`
Mutex / shared state guard	`asyncio.Lock()`	no built-in — use a promise chain or library
Async memoize	manual with `dict` + `asyncio.Event`	manual with `Map` + stored `Promise`
Catch async error	`try/except`	`try/catch` (same**

Bridging the Asyncio/TS gap

Python version of first to succeed

```
import asyncio
```

```python
    async def first_success(coros):
        tasks = [asyncio.create_task(c) for c in coros]
        errors = []
        try:
            for fut in asyncio.as_completed(tasks):
                try:
                    return await fut
                except Exception as e:
                    errors.append(e)
            raise ExceptionGroup("all failed", errors)
        finally:
            for t in tasks:
                t.cancel()
```

`as_completed` yields futures in completion order, so you're always trying the next thing that finished. The finally cancels whatever's still pending once you return — same cleanup Promise.any does internally.

Usage mirrors the JS exactly:

```
result = await first_success([fetch_from_a(), fetch_from_b(), fetch_from_c()])
```

The main differences from Promise.any: - `ExceptionGroup`(Python 3.11+) vs `AggregateError` — same idea, different name - Explicit cancellation — JS promises can't be cancelled, Python tasks can and should be - `asyncio.create_task` is the critical call — without it, the coroutines don't start running until you await them, which would serialize rather than parallelize

Call order vs completion order: `Promise.all` returns results in the order the promises were created, regardless of which resolved first. There is no `as_completed` in TS — to get completion-order results, wire each promise to push into a shared array as it resolves.

```typescript
// completion order
const results: Result[] = [];
await Promise.all(
  items.map(async (item) => {
    const r = await fetch(item);
    results.push(r);        // pushed as each resolves
  })
);
```

Bounded concurrency: Python has `asyncio.Semaphore` as a first-class primitive. In TS you implement it yourself — track an in-flight counter and queue work until a slot frees up. Common interview pattern.

```typescript
async function withConcurrency<T>(
  items: T[],
  maxConcurrent: number,
  fn: (item: T) => Promise<void>,
): Promise<void> {
  const queue = [...items];
  // we don't need to IIFE because `Array.from` maps over indices,
  // so each async function is called on the spot, and workers
```

```
  // is an array of live running Promises, not of defined functions
  const workers = Array.from({ length: maxConcurrent }, async () => {
    // no race condition possible here; JS is single-threaded
    // `while` check and `shift()` happen in the same burst with no `await`
    // so there's no interleaving possible
    while (queue.length > 0) {
      await fn(queue.shift()!);
    }
  });
  await Promise.all(workers);
}
```

Async generators: `for await (const x of y)` is the TS equivalent of `async for`. Use `async function*` to produce a lazy async stream — useful when you don't want to buffer all results in memory before returning.

```
async function* paginate(url: string): AsyncIterable<Item[]> {
  let cursor: string | null = url;
  while (cursor) {
    const page = await fetch(cursor).then(r => r.json());
    yield page.items;
    cursor = page.nextCursor ?? null;
  }
}

for await (const page of paginate("/api/items")) {
  process(page);
}
```

Async memoize / dedup: store the `Promise` itself in the cache, not the resolved value. Concurrent callers awaiting the same key all attach to the same promise — `fn` is called exactly once.

```
const cache = new Map<string, Promise<Result>>();

function memoize(key: string, fn: () => Promise<Result>): Promise<Result> {
  if (!cache.has(key)) cache.set(key, fn());
  return cache.get(key)!;
}
```

No built-in mutex: unlike Python's `asyncio.Lock`, TS has nothing equivalent in stdlib. For single-threaded async code a flag or promise chain is usually enough; for worker threads you need `Atomics` or a library.

```
// promise-chain mutex — each caller chains onto the previous
class Mutex {
  private lock = Promise.resolve();
  acquire<T>(fn: () => Promise<T>): Promise<T> {
    const next = this.lock.then(() => fn());
    this.lock = next.then(() => {}, () => {});
```

```
    return next;
  }
}
```

Retry: attempt up to `maxAttempts` times, re-raise on final failure.

```python
# asyncio
async def with_retry(fn, max_attempts: int):
    for attempt in range(1, max_attempts + 1):
        try:
            return await fn()
        except Exception:
            if attempt == max_attempts:
                raise
```

```typescript
// TypeScript
async function withRetry<T>(fn: () => Promise<T>, maxAttempts: number): Promise<T> {
  for (let attempt = 1; attempt <= maxAttempts; attempt++) {
    try { return await fn(); }
    catch (e) { if (attempt === maxAttempts) throw e; }
  }
  throw new Error("unreachable");  // satisfies tsc — loop always throws or returns
}
```

Fan-out with per-item retry (concurrent messages, sequential retries per message):

```python
# asyncio
results = await asyncio.gather(
    *[with_retry(lambda: handler(msg), max_attempts) for msg in messages],
    return_exceptions=True,
)
```

```typescript
// TypeScript
await Promise.all(messages.map(msg =>
  withRetry(() => handler(msg), maxAttempts)
));
```

Gotchas

`==` vs `===` Always use `===`. The `==` does type coercion (`0 == ""` is `true`).

`for ... of` vs `for ... in`

```
for (const x of arr) { }   // like Python's for x in lst — use this
for (const k in obj) { }   // iterates keys of an object — rarely what you want
```

Never use `for ... in` on arrays.

When to use `for ... of`?

For plain a object (not a Map) with dynamic keys, typically JSON received from an API: `const obj = { a: 1, b: 2, c: 3 }; for (const key in obj) { console.log(key, obj[key]); }`

Modern alternative that most TS devs prefer:

```
Object.keys(obj)    // ["a", "b", "c"]
Object.values(obj)  // [1, 2, 3]
Object.entries(obj) // [["a", 1], ["b", 2], ["c", 3]]
```

Then we can .map(), .filter() etc. on them like normal arrays.

`for ... in` has a gotcha — it also iterates inherited properties from the prototype chain. If using it, guard with:

```
if (Object.hasOwn(obj, key)) { ... }
```

In practice: if you control the data structure, use `Map` and `for ... of`. If you're dealing with plain objects from JSON, use `Object.entries()`.
—

`this` loses context

```
class Foo {
  name = "foo";
  greet() { console.log(this.name); }
}
const f = new Foo();
const g = f.greet;
g();  // 'this' is undefined — crashes
```

In Python, bound methods carry `self`. In TS, `this` depends on *how* the function is called. Use arrow functions when passing methods as callbacks:

```
bus.subscribe("X", (e) => this.handle(e));   // safe
bus.subscribe("X", this.handle);             // risky, might be unbound!
```

Integers don't exist Everything is `number` (float). `3 / 2` is `1.5`, not `1`. Use `Math.floor(3 / 2)` for integer division.

`undefined` and `null` are different - Missing object property → `undefined` - Explicit "no value" → `null` - Both are falsy. `x == null` catches both; `x === null` catches only `null`.

Empty objects (including array, map, sets) are truthy

```
if (arr.length > 0) { }  // correct
if (arr) { }             // always true, even for []
```

Value	Truthy/Falsy
new Map()	truthy
new Set()	truthy
[]	truthy
{}	truthy
“”	falsy
0	falsy
null	falsy
undefined	falsy

Empty string is falsy — same as Python. But all empty containers are truthy, unlike Python where bool([]) is False.

Rule of thumb: in TS, only primitives can be falsy. Objects (including arrays, maps, sets) are always truthy.

Array spread vs copy

```
const b = [...a];   // shallow copy, like a[:]
const b = a;        // same reference — mutations affect both!
```

NaN ≠= NaN

```
Number("oops")    // NaN
NaN === NaN       // false (!)
Number.isNaN(x)   // correct way to check
```

Default sort coerces to strings

```
[10, 9, 2].sort()             // [10, 2, 9] — wrong!
[10, 9, 2].sort((a, b) => a - b)  // [2, 9, 10] — correct
```

Always pass a comparator for numbers.

`Array.from` calls its factory immediately — results are live Promises

```
// workers is an array of return values — live Promises when factory is async
const workers = Array.from({ length: n }, async () => { ... });

// contrast: workerFns is an array of dormant functions — nothing runs yet
const workerFns = Array.from({ length: n }, () => async () => { ... });
const workers = workerFns.map(fn => fn());  // now they start
```

The single-arrow form `async () =>` is called by `Array.from` on the spot. The double-arrow form `() => async () =>` returns a function — you need the extra `fn()` call to start it. Easy to mix up when building worker pools.

Testing

it.each / parametrize

```python
# pytest
@pytest.mark.parametrize("a,b,expected", [
    (1, 2, 3),
    (0, 0, 0),
])
def test_add(a, b, expected):
    assert add(a, b) == expected

# async
@pytest.mark.asyncio
@pytest.mark.parametrize("id,expected", [("m1", 42)])
async def test_fetch(id, expected):
    result = await fetch(id)
    assert result.kwh == expected
```

```javascript
// vitest
it.each([
  [1, 2, 3],
  [0, 0, 0],
])("add(%i, %i) = %i", (a, b, expected) => {
  expect(add(a, b)).toBe(expected);
});

// named tuples (cleaner for many fields)
it.each([
  { id: "m1", expected: 42 },
])("fetches $id", async ({ id, expected }) => {
  const result = await fetch(id);
  expect(result.kwh).toBe(expected);
});
```

`it.each` with named objects: use `$fieldName` in the test name string. `it.each` with arrays: use `%i` (number), `%s` (string), `%o` (object).

Sync vs async assertions

```python
# sync — plain assert
assert result == expected
```

```python
# async — just await in the test body
result = await fetch(id)
assert result.kwh == 42

# asserting an exception
with pytest.raises(ValueError, match="boom"):
    risky()

# async exception
with pytest.raises(ValueError):
    await async_risky()
```

```javascript
// sync
expect(result).toBe(expected);          // primitive equality
expect(result).toEqual(expected);       // deep equality (like == in Python)
expect(result).toStrictEqual(expected); // deep + same type (no extra keys)

// async — just await in the test body (test fn is async)
const result = await fetch(id);
expect(result.kwh).toBe(42);

// asserting a thrown exception
expect(() => risky()).toThrow("boom");

// async exception
await expect(asyncRisky()).rejects.toThrow("boom");
// note: no await on asyncRisky() itself — pass the Promise to expect()
```

Key difference: in vitest you pass the *Promise* (not the awaited result) to `expect( … ).rejects`. In pytest you just `await` inside the `with` block.

Mocking

Operation	unittest.mock	mockito (Python)	vitest
Create mock (sync)	`MagicMock(return_value=42)`	`when(mod).fn( … ).thenReturn(42)`	`vi.fn().mockReturnValue(42)`
Create mock (async)	`AsyncMock(return_value=42)`	`when(mod).fn( … ).thenReturn(42)`	`vi.fn().mockResolvedValue(42)`
Mock raises	`MagicMock(side_effect=ValueError())`	`when(mod).fn( … ).thenRaise(err)`	`vi.fn().mockRejectedValue(err)`
Spy real method	`patch.object(obj, "m", wraps=obj.m)`	`spy2(obj.method)`	`vi.spyOn(obj, "method")`
Called once with args	`m.assert_called_once_with("m1")`	`verify(mod).fn("m1")`	`expect(m).toHaveBeenCalledWith("m1")`
Called N times	`assert m.call_count == 3`	`verify(mod, times(3)).fn( … )`	`expect(m).toHaveBeenCalledTimes(3)`
Never called	`m.assert_not_called()`	`verify(mod, never()).fn( … )`	`expect(m).not.toHaveBeenCalled()`
Last call args	`m.assert_called_with("m1")`	—	`expect(m).toHaveBeenLastCalledWith("m1"`

Operation	unittest.mock	mockito (Python)	vitest
Nth call args	`m.call_args_list[n].args`	—	`expect(m).toHaveBeenNthCalledWith(n, ...)`
Reset history	`m.reset_mock()`	`unstub()`	`vi.clearAllMocks()`
Reset + implementation	—	—	`vi.resetAllMocks()`
Restore spy	context manager exit	`unstub()`	`spy.mockRestore()`

Inspecting raw call args (when table methods aren't specific enough):

```python
mock.call_args_list[0].args[0]   # first arg of first call
mock.call_args.args[0]           # first arg of most recent call
```

```typescript
mock.mock.calls[0][0]    // first arg of first call
mock.mock.calls.at(-1)   // all args of last call
```

Common helpers everyone eventually writes

Entity factory — avoids brittle fixtures by building objects with sensible defaults and accepting overrides. The most universally written helper.

```python
# pytest
def make_meter_reading(**overrides) -> MeterReading:
    return MeterReading(meter_id="m1", kwh=42.0, timestamp=datetime.now(), **overrides)

# usage
make_meter_reading(kwh=0)   # only override what matters for this test
```

```typescript
// vitest
const makeMeterReading = (overrides: Partial<MeterReading> = {}): MeterReading => ({
  meterId: "m1", kwh: 42, timestamp: new Date(), ...overrides,
});
```

Nth call inspector — cleaner than indexing `.call_args_list` / `.mock.calls` inline:

```python
def nth_call(mock, n: int, arg: int = 0):
    return mock.call_args_list[n].args[arg]

nth_call(mock_fetch, 0)      # first arg of first call
nth_call(mock_fetch, -1, 1)  # second arg of last call
```

```typescript
const nthCall = (mock: ReturnType<typeof vi.fn>, n: number, arg = 0) =>
  mock.mock.calls.at(n)?.[arg];

nthCall(mockFetch, 0)     // first arg of first call
nthCall(mockFetch, -1, 1) // second arg of last call
```

Mock factory — centralise default mock behaviour so tests only specify what they care about:

```python
def make_mock_fetcher(return_value=None, side_effect=None):
    m = AsyncMock()
    if side_effect: m.side_effect = side_effect
    else: m.return_value = return_value or make_meter_reading()
    return m
```

```typescript
const makeMockFetcher = (overrides?: Partial<MeterReading>) =>
  vi.fn().mockResolvedValue(makeMeterReading(overrides));
```

Flush promises — let all pending microtasks/macrotasks settle without real timers:

```python
# pytest-asyncio handles this automatically — just await
await asyncio.sleep(0)   # yield control once to event loop if needed
```

```typescript
const flushPromises = () => new Promise(r => setTimeout(r, 0));
await flushPromises();   // lets queued microtasks resolve
// or use vitest's built-in:
await vi.runAllTimersAsync();   // requires vi.useFakeTimers()
```

Approximate equality — for floats and timestamps:

```python
assert result.kwh == pytest.approx(42.0, rel=1e-3)
assert abs((result.timestamp - expected).total_seconds()) < 1
```

```typescript
expect(result.kwh).toBeCloseTo(42.0, 3);          // 3 decimal places
expect(result.timestamp.getTime()).toBeCloseTo(    // within 1 second
  expected.getTime(), -3);
// or just:
expect(Math.abs(result.timestamp.getTime() - expected.getTime())).toBeLessThan(1000);
```

Partial object matching — assert on the fields you care about:

```python
assert result == pytest.approx({"kwh": 42})       # partial dict
# or with dataclasses:
assert result.kwh == 42 and result.meterId == "m1"
```

```typescript
expect(result).toMatchObject({ kwh: 42, meterId: "m1" });  // ignores extra fields
expect(results).toEqual(expect.arrayContaining([
  expect.objectContaining({ meterId: "m1" }),
]));
```

Source: https://gist.github.com/candeira/7d7a2e8a581607eaa9a591f138593631