Collections and Generics

Typing Lists, Dicts, and Custom Containers

Effective Programming Practices for Economists

Built-in Collection Types

```
names: list[str] = ["Alice", "Bob"]
scores: dict[str, int] = {"Alice": 95, "Bob": 87}
coordinates: tuple[float, float] = (1.0, 2.0)
unique_ids: set[int] = {1, 2, 3}
```

Pattern: `container[element_type]`

- `list[T]` - list of T
- `dict[K, V]` - dict with keys K and values V
- `tuple[T1, T2]` - tuple with specific types
- `set[T]` - set of T

Lists

```
# List of integers
numbers: list[int] = [1, 2, 3, 4, 5]

# List of strings
names: list[str] = ["Alice", "Bob", "Carol"]

# Nested list (matrix)
matrix: list[list[float]] = [
    [1.0, 2.0],
    [3.0, 4.0],
]

# Empty list with type annotation
results: list[float] = []
```

Dictionaries

```python
# String keys, integer values
word_counts: dict[str, int] = {"hello": 5, "world": 3}

# Integer keys, list values
user_scores: dict[int, list[float]] = {
    1: [95.0, 87.5],
    2: [78.0, 82.0],
}

# Nested dictionaries
config: dict[str, dict[str, str]] = {
    "database": {"host": "localhost", "port": "5432"},
    "cache": {"host": "redis", "port": "6379"},
}
```

Tuples: Fixed vs Variable Length

Fixed-length tuple (specific types per position):

```python
# Exactly two elements: (name, age)
person: tuple[str, int] = ("Alice", 30)

# Exactly three elements
rgb: tuple[int, int, int] = (255, 128, 0)
```

Variable-length tuple (homogeneous):

```python
# Any number of integers
numbers: tuple[int, ...] = (1, 2, 3, 4, 5)
```

Sets and Frozensets

```python
# Mutable set
active_users: set[int] = {1, 2, 3}

# Immutable frozenset
valid_statuses: frozenset[str] = frozenset({"active", "pending", "closed"})
```

When to use which:

- `set` - when you need to add/remove elements
- `frozenset` - when you need a hashable set (dict keys, set of sets)

Abstract Collection Types

```python
from collections.abc import Sequence, Mapping, Iterable

def process_items(items: Sequence[int]) -> int:
    """Accept list, tuple, or any sequence."""
    return sum(items)

def lookup(data: Mapping[str, int], key: str) -> int:
    """Accept dict or any mapping."""
    return data[key]

def count_items(items: Iterable[str]) -> int:
    """Accept any iterable (list, set, generator, etc.)."""
    return sum(1 for _ in items)
```

When to Use Abstract Types

Use concrete types (`list` , `dict`) when:

- You need specific methods (e.g., `list.append`)
- You’re returning a value (be specific)

Use abstract types (`Sequence` , `Mapping`) when:

- You only read from the collection
- You want to accept multiple input types
- You’re writing library code

```python
# Good: Accept any sequence, return specific list
def double_values(values: Sequence[int]) -> list[int]:
    return [v * 2 for v in values]
```

Generic Functions

```python
# Type parameter in brackets after function name
def first[T](items: list[T]) -> T:
    """Return first element, preserving the type."""
    return items[0]

# Usage:
first([1, 2, 3])       # Returns int
first(["a", "b"])      # Returns str
```

`T` is a type parameter

- Same `T` means "same type throughout"
- Type checker infers T from arguments

Constrained Type Parameters

```
# T can only be int or float
def add_numbers[T: (int, float)](a: T, b: T) -> T:
    return a + b

add_numbers(1, 2)       # OK: both int
add_numbers(1.0, 2.0)   # OK: both float
add_numbers(1, 2.0)     # Error: mixing types!
```

Constraint syntax: `T: (Type1, Type2)`

Bounded Type Parameters

```
class Animal:
    def speak(self) -> str:
        return "..."

class Dog(Animal):
    def speak(self) -> str:
        return "Woof!"

# T must be Animal or a subclass
def make_speak[T: Animal](animal: T) -> str:
    return animal.speak()
```

Bound syntax: `T: BaseType`

Generic Classes

```python
class Stack[T]:
    def __init__(self) -> None:
        self._items: list[T] = []

    def push(self, item: T) -> None:
        self._items.append(item)

    def pop(self) -> T:
        return self._items.pop()

# Usage:
int_stack: Stack[int] = Stack()
int_stack.push(1)
int_stack.push(2)
value: int = int_stack.pop()  # Type is preserved
```

Multiple Type Parameters

```python
class Pair[K, V]:
    def __init__(self, key: K, value: V) -> None:
        self.key = key
        self.value = value

    def swap(self) -> "Pair[V, K]":
        return Pair(self.value, self.key)

# Usage:
p: Pair[str, int] = Pair("age", 30)
swapped: Pair[int, str] = p.swap()
```

Callable Types

```python
from collections.abc import Callable

# Function that takes two ints and returns a float
Operation = Callable[[int, int], float]

def apply_operation(
    a: int,
    b: int,
    op: Callable[[int, int], float],
) -> float:
    return op(a, b)

def divide(x: int, y: int) -> float:
    return x / y

result = apply_operation(10, 3, divide)
```

Generic Callable

```python
from collections.abc import Callable

def apply_twice[T, R](
    func: Callable[[T], R],
    value: T,
) -> tuple[R, R]:
    """Apply function twice, return both results."""
    return func(value), func(value)

def square(x: int) -> int:
    return x * x

result: tuple[int, int] = apply_twice(square, 5)
```

Common Patterns

Optional list parameter:

```
def process(items: list[int] | None = None) -> list[int]:
    if items is None:
        items = []
    return [x * 2 for x in items]
```

Dict with default:

```
def get_config(
    overrides: dict[str, str] | None = None,
) -> dict[str, str]:
    config = {"host": "localhost", "port": "8080"}
    if overrides:
        config.update(overrides)
    return config
```

Type Narrowing

```python
def process(value: int | str | None) -> str:
    if value is None:
        return "empty"
    if isinstance(value, int):
        # Here, type checker knows value is int
        return str(value * 2)
    # Here, type checker knows value is str
    return value.upper()
```

Type checkers understand:

- `if x is None` / `if x is not None`
- `isinstance(x, Type)`
- `assert isinstance(x, Type)`

Summary

Collections:

- `list[T]`, `dict[K, V]`, `tuple[T1, T2]`, `set[T]`
- Use `Sequence`, `Mapping`, `Iterable` for flexible inputs

Generics:

- `def func[T](...)` for generic functions
- `class Foo[T]:` for generic classes
- `T: BaseType` for bounds, `T: (A, B)` for constraints