# YZM1022

## Advanced Programming

### Week 12: Generic Programming and Type Systems

**Instructor:** Ekrem Çetinkaya

**Date:** 13.05.2026

# Recap - Week 11: Recursion & Dynamic Programming

**Recursion Patterns**
Five shapes: linear, divide & conquer, accumulator, multiple, mutual. Each reduces a problem to a smaller version of itself.

**Advanced Applications**
Tower of Hanoi ($2^n$–1 optimal moves), merge sort and quicksort (O(n log n) divide & conquer), backtracking (try/recurse/undo for N-queens and permutations).

**Recursion Limits**
Python caps the call stack at 1000 frames. Deep recursion -> convert to iteration with an explicit stack.

**Dynamic Programming**
Memoization (top-down, `@lru_cache`) and tabulation (bottom-up table) both solve overlapping subproblems in O(n) instead of O($2^n$). Classic problems: Fibonacci, coin change, LCS, knapsack.

# Today's Agenda

**Type Hints**

Annotations for variables, functions, collections, Optional/Union, Literal, type aliases

**Generics**

TypeVar · Generic classes · Bounded and constrained type variables

**Advanced Typing**

Callable · ParamSpec · TypedDict · mypy

# Dynamic Typing - Flexibility vs Safety

Python is a **dynamically typed** language meaning types are determined and checked when each line executes, not before the program starts.

- This gives Python its famous flexibility: the same variable can hold an integer, then a string, then a list, and Python won't complain.
- The downside is that the interpreter has no way to detect a type mismatch until that specific line actually runs, which might be milliseconds after deployment or weeks later on a rare code path that only triggers with unusual input.

```
def apply_discount(price, discount):
    return price - (price * discount)

apply_discount(100, 0.1)    # 90.0  - correct
apply_discount("100", 0.1)  # TypeError at runtime
apply_discount(100, 10)     # -900.0 - logically wrong, no error raised!
apply_discount(None, 0.1)   # TypeError: unsupported operand type

def get_user_name(users, user_id):
    return users[user_id]["name"]  # What if users[user_id] is None?
                                   # What if "name" key is missing?
```

All of these look syntactically fine. Python never warns you until the line executes.

# Dynamic Typing - The Advantages

Dynamic typing is not a flaw it's a design choice that makes Python fast to write and expressive for exploratory work.

- When you don't have to declare types upfront, you can focus on logic rather than satisfying a compiler, which is why Python dominates data science, rapid prototyping, and scripting.
- The same function works for any type automatically, and reshaping heterogeneous data from JSON or APIs requires no ceremony.

```
# One function, any type — no generics needed
def first(items):
    return items[0]

first([1, 2, 3])              # 1
first(["a", "b"])             # "a"
first([(1, "x"), (2, "y")])   # (1, "x")

# Rapid prototyping — no declarations needed
config = {}
config["retries"] = 3
config["timeout"] = 5.0
config["host"] = "api.example.com"   # Mix types freely

# Great for exploratory data work
data = json.loads(response.text)     # Just use it, no schema required
print(data["users"][0]["name"])
```

# Typed Languages - Catching This Before It Ships

In languages like TypeScript, Java, Kotlin, and Rust, type checking is a mandatory compilation step.

- The compiler analyzes every function call and verifies that the argument types match the declared parameter types before producing any executable.
- If a mismatch is found, compilation fails with a clear error pointing to the offending line, and no program is produced at all.
- This means an entire category of bugs (wrong argument types, calling methods that don't exist on a value, forgetting to handle `None`) are caught during development on the developer's machine rather than discovered in production when a user triggers an edge case.

### TypeScript

```typescript
function applyDiscount(price: number, discount: number): number {
  return price - price * discount;
}

applyDiscount("100", 0.1);
// Compile Error:
// Argument of type 'string' is not
// assignable to type 'number'
```

### Java

```java
public double applyDiscount(
        double price, double discount) {
    return price - (price * discount);
}

applyDiscount("100", 0.1);
// Compile Error:
// incompatible types: String cannot
// be converted to double
```

# Python Type Hints

# Why Type Hints?

Without annotations, a function's contract is invisible and we must read the body to know what types it expects or returns. Type hints surface that contract in the signature, where tools can verify it and teammates can read it at a glance.

```
# Without type hints — what types are expected?
def process_data(data, threshold):
    results = []
    for item in data:
        if item > threshold:          # Does > work? Depends on item type
            results.append(item * 2) # Does * 2 work? Depends on item type
    return results                    # What does this return?

# Same bug as apply_discount earlier: Python won't stop you from calling:
process_data("hello world", 5)      # Iterates characters — silently wrong
process_data([1, 2, 3], "high")     # 💥 TypeError on >

# With type hints — the mistake is caught before the code runs:
def process_data(data: list[int], threshold: int) -> list[int]:
    results: list[int] = []
    for item in data:
        if item > threshold:
            results.append(item * 2)
    return results

process_data("hello world", 5)
```

# Benefits of Type Hints

## For Developers

### Better IDE Support

- Autocomplete knows the types
- Refactoring is safer
- Navigation is easier

### Self-Documenting Code

- Types explain intent
- No need to read implementation
- API is clear

### Catch Bugs Early

- Static analysis finds errors
- Before code even runs
- Prevents runtime surprises

## For Teams

### Clearer Contracts

- Function signatures are explicit
- Less guessing about interfaces
- Easier code reviews

### Easier Refactoring

- Type checker validates changes
- Confidence in modifications
- Safe at scale

### Better Documentation

- Types are always up-to-date
- Unlike comments/docstrings
- Tooling can verify

# Basic Type Annotations

Python's annotation syntax places a type after a colon for variables and parameters (`name: str`, `age: int = 25`) and after an arrow for return values (`-> bool`).

- These annotations are **never read or enforced at runtime** - Python executes `add("hello", "world")` without raising an error because the interpreter doesn't consult type annotations during execution.
- The annotations exist purely for static analysis tools like mypy and for IDE features like autocomplete; to actually catch type errors we must run the type checker as a separate step.

# Basic Type Annotations

```python
# Variable annotations
name: str = "Alice"
age: int = 25
price: float = 19.99
is_active: bool = True

# Function annotations
def greet(name: str) -> str:
    return f"Hello, {name}!"

def add(a: int, b: int) -> int:
    return a + b

def is_adult(age: int) -> bool:
    return age >= 18

# None return type
def print_message(message: str) -> None:
    print(message)

# Type hints are not enforced at runtime and this will not raise an error:
result = add("hello", "world")  # Returns "helloworld
```

# Collection Types

When annotating a collection, we must specify not just the container type but also the element type.

- `list[int]` tells that every element is an integer, not merely that the variable is some kind of list.
- Python 3.9 made this convenient by allowing built-in types to be used directly as generics (`list[int]`, `dict[str, int]`), eliminating the need to import `List` and `Dict` from the `typing` module that earlier Python versions required.
- Tuples have special semantics: `tuple[int, str]` means exactly two elements of those specific types in that order, while `tuple[int, ...]` with the ellipsis means a variable-length homogeneous tuple where every element is an integer.

# Collection Types

```python
# Modern syntax (Python 3.9+)
numbers: list[int] = [1, 2, 3]
names: set[str] = {"Alice", "Bob"}
scores: dict[str, int] = {"Alice": 95, "Bob": 87}
point: tuple[int, int] = (10, 20)
coordinates: tuple[float, float, float] = (1.0, 2.0, 3.0)

# Variable-length tuple
values: tuple[int, ...] = (1, 2, 3, 4, 5)

# Nested collections
matrix: list[list[int]] = [[1, 2], [3, 4]]
user_tags: dict[str, list[str]] = {
    "Alice": ["python", "java"],
    "Bob": ["javascript"]
}

# For Python 3.8 and earlier, use typing module
from typing import List, Dict, Set, Tuple

numbers: List[int] = [1, 2, 3]
scores: Dict[str, int] = {"Alice": 95}
```

# Optional and Union

Many functions can fail to find what they're looking for

- A database query might return no row, a dictionary's `.get()` returns `None` for missing keys.
- `Optional[str]` (shorthand for `str | None`) makes this possibility explicit in the type signature, forcing callers to acknowledge that the function might return `None` rather than silently assuming it always succeeds.
- `Union[int, str]` covers a different scenario: the value is always present, but it could be either of two types
  - Common when parsing external data where numbers might come in as strings.
- Python 3.10 introduced the `|` pipe operator as a cleaner replacement for both forms, so modern code writes `str | None` and `int | str` directly without importing anything from `typing`.

# Optional and Union

```python
from typing import Optional, Union

# Optional = can be the type OR None
def find_user(user_id: int) -> Optional[str]:
    """Returns username or None if not found"""
    users = {1: "Alice", 2: "Bob"}
    return users.get(user_id)  # Returns str or None

# Union = can be any of the listed types
def process(value: Union[int, str]) -> str:
    if isinstance(value, int):
        return f"Number: {value}"
    return f"String: {value}"

# Python 3.10+ syntax: use | operator
def find_user(user_id: int) -> str | None:
    pass

def process(value: int | str) -> str:
    pass
```

# Any and Unknown Types

`Any` is the complete escape hatch from the type system.

- Once a value is typed as `Any`, Python stops verifying anything you do with it and the checker stays silent.
- The critical difference from `object` is that `object` is the actual root of Python's class hierarchy, so it's a valid and strict type that prevents you from calling methods the checker can't verify.
- `Any` pretends to be every type simultaneously, disabling all verification rather than enforcing a stricter baseline.

**When to use Any:**

- Interfacing with untyped code
- Complex dynamic behavior
- Gradual typing migration

# Any and Unknown Types

```python
from typing import Any

# Any: Opt out of type checking
def log_anything(value: Any) -> None:
    print(value)  # No type checking on value

log_anything(42)
log_anything("hello")
log_anything([1, 2, 3])

# object: Base of all types (stricter than Any)
def process_object(obj: object) -> str:
    # Can't do much with object, need to check type
    return str(obj)
```

# The `Any` Problem

The main problem is `Any` is contagious.

- Once one function returns `Any`, every downstream function that receives its output is forced to accept `Any` as a parameter type, and its own return type often becomes `Any` as well, silencing the checker across the entire call chain.
- This propagation effect is sometimes called the **any virus**: a single poorly-typed boundary function can strip type safety from everything built on top of it, leaving annotations that exist on paper but provide no real guarantees.

```
# One function returning Any poisons everything downstream:
def load_config(path: str) -> Any:       # Returns Any
    return json.load(open(path))

def get_timeout(config: Any) -> Any:     # Must accept Any, returns Any
    return config["timeout"]

def make_request(url: str, timeout: Any) -> Any:  # timeout is Any too
    ...
# Gradual typing done wrong:
result: Any = make_request("https://api.example.com", get_timeout(load_config("cfg.json")))
result.non_existent_method()  # checker stays silent - runtime crash waiting to happen

# Use specific types, or at minimum, cast at the boundary:
config: dict[str, int | str] = cast(dict[str, int | str], load_config("cfg.json"))
```

# Literal and Final

`Literal` tightens the type system from validating the shape of a value to validating its exact content.

- `status: Literal["pending", "active", "done"]` is far stronger than `status: str` because checker will reject any string that isn't one of those three values, catching invalid states at check time.
- This is the lightweight alternative to defining a full `Enum` class when you need a fixed set of allowed values for things like HTTP methods, operation modes, or status flags.
- `Final` serves a different purpose: it marks a variable as a write-once constant, so the type checker raises an error if any code later attempts to reassign it, making configuration values and module-level constants self-documenting and protected from accidental mutation.

**Use Literal for:**

- String enums without defining Enum class
- Specific allowed values
- Mode/flag parameters

# Literal and Final

```python
from typing import Literal, Final

# Literal: Restrict to specific values
def set_status(status: Literal["pending", "active", "done"]) -> None:
    print(f"Status: {status}")

set_status("active")   # OK
set_status("invalid")  # Type error!

# Final: Constant that shouldn't be reassigned
MAX_RETRIES: Final[int] = 3
API_URL: Final[str] = "https://api.example.com"

MAX_RETRIES = 5  # Type error: cannot assign to Final!

# Final class attribute
class Config:
    DEBUG: Final[bool] = False
```

# Type Aliases

As codebases grow, complex type expressions like `dict[str, int | str | None]` get copy-pasted across dozens of function signatures, making the code hard to read and even harder to change consistently when the data model evolves.

- Type aliases solve this by binding the expression to a meaningful name
  - *e.g.*, `UserData: TypeAlias = dict[str, int | str | None]`
- Function signatures read like domain vocabulary rather than implementation details.
- The explicit `TypeAlias` annotation (introduced in Python 3.10) is important because without it, a bare assignment like `UserData = dict[str, int | str | None]` looks identical to a regular variable declaration, and some tools may not recognize it as an alias.

# Type Aliases

```python
from typing import TypeAlias, Callable

# Create readable aliases for complex types
UserId = int
Username = str
UserData: TypeAlias = dict[str, int | str | None]

def get_user(user_id: UserId) -> UserData:
    return {"id": user_id, "name": "Alice", "age": 30}

# Complex nested types become readable
JSON: TypeAlias = dict[str, "JSON"] | list["JSON"] | str | int | float | bool | None

# Callback type alias
Callback: TypeAlias = Callable[[int, str], None]
```

# Type Aliases - Clean Signatures

The real advantage of type aliases appears at function boundaries.

- Function signature is the first thing a collaborator or API consumer reads, and hiding it in nested generics (`dict[str, dict[str, int | str | None]]`) signals complexity rather than intent.
- Replacing those raw expressions with semantic names makes signatures read like a description of the problem domain, communicating what the data represents rather than how it is stored.
- Type aliases also make refactoring safer: if the shape of `UserData` ever changes, you update one definition and the type checker automatically validates every usage site across the entire codebase.

```
# Without aliases - hard to read
def complex_function(
    users: dict[str, dict[str, int | str | None]],
    callback: Callable[[dict[str, int | str | None]], bool]
) -> list[dict[str, int | str | None]]:
    pass

# With aliases - much cleaner
def complex_function(
    users: dict[str, UserData],
    callback: Callable[[UserData], bool]
) -> list[UserData]:
    pass
```

# From Naming Types to Parameterizing Them

Type aliases give us readable names for complex concrete types ( `UserData` , `Callback` , `JSON` ), but they still describe *specific* types.

**The remaining problem:**

```
def first_int(items: list[int]) -> int: ...
def first_str(items: list[str]) -> str: ...
def first_float(items: list[float]) -> float: ...
```

Same logic, three functions. Using `Any` collapses them into one but loses type safety and the return type becomes unknown.

**What Generics add:**

The *type itself* becomes a parameter, `T` . The function works for any type, and the return type is whatever the caller passes in.

```
def first(items: list[T]) -> T: ...

first([1, 2, 3])      # -> int
first(["a", "b"])     # -> str
```

Write once. Type-safe for every caller.

# Generics

# The Need for Generics

Writing one function per type ( `first_int` , `first_str` , `first_float` ) violates DRY and becomes unmanageable as the number of types grows.

- Xix types means six nearly identical functions, and a bug fix must be applied in six places.
- The tempting shortcut is to use `Any` , which collapses them into one function, but it collapses all type information too:
    - The return type becomes `Any` , mypy can no longer verify callers, and autocomplete in the IDE disappears.

Generics provide the third option: write the function once, parameterize it by a type variable `T` , and let the type checker substitute the correct concrete type at each call site,achieving both reuse and full type safety simultaneously.

# The Need for Generics

```python
# Problem: Writing type-safe reusable code

def first_int(items: list[int]) -> int:
    return items[0]

def first_str(items: list[str]) -> str:
    return items[0]

def first_float(items: list[float]) -> float:
    return items[0]

# Same logic, different types - not DRY!

# Solution 1: Use Any (loses type safety)
from typing import Any

def first_any(items: list[Any]) -> Any:
    return items[0]

result = first_any([1, 2, 3])  # result is Any, not int!
result + 1  # No type checking - could be wrong

# Solution 2: Use Generics!
```

# TypeVar - Type Variables

A `TypeVar` is a symbolic placeholder for a concrete type that will be determined at each call site.

- `T = TypeVar('T')` doesn't mean 'any type is allowed' (as `Any` does), it means '*whatever type the caller provides for the input, the output will be that same type*.'
- When the type checker sees `first([1, 2, 3])`, it substitutes `int` for `T` and infers that the return is `int`
  - For `first(['a', 'b'])` it substitutes `str`.
- Unlike `Any`, the TypeVar preserves the input-output relationship, so calling `.upper()` on the result of `first([1, 2, 3])` is correctly flagged as a type error.

# TypeVar - Type Variables

```python
from typing import TypeVar

# Define a type variable
T = TypeVar('T')

# Now T can be any type
def first(items: list[T]) -> T:
    """Return first item, preserving type information"""
    return items[0]

# Type checker infers T based on usage
int_result = first([1, 2, 3])        # T = int, returns int
str_result = first(["a", "b", "c"])  # T = str, returns str

# Type safety preserved!
int_result + 1      # OK: int + int
str_result.upper()  # OK: str method

# Type error caught:
# int_result.upper()  # Error: int has no 'upper'
```

# Multiple Type Variables

When a function involves two independently varying types, such as swapping a pair or combining two lists element-by-element, a single TypeVar is insufficient because it would force both types to be the same.

- Two separate TypeVars `T` and `U` let the checker track each one independently:
  - For `swap((1, 'hello'))`, it infers `T=int` and `U=str`, then correctly types the result as `tuple[str, int]`.
- The `zip_with` example is with three TypeVars: `T` and `U` for the input lists, and `V` for the result type produced by combining them
  - The checker verifies that the function argument produces `V` from `T` and `U`, and that the output list is `list[V]`.

# Multiple Type Variables

```python
from typing import TypeVar, Callable

T = TypeVar('T')
U = TypeVar('U')
V = TypeVar('V')

def swap(pair: tuple[T, U]) -> tuple[U, T]:
    return (pair[1], pair[0])

result = swap((1, "hello"))  # Returns ("hello", 1)

def zip_with(
    func: Callable[[T, U], V],
    list1: list[T],
    list2: list[U]
) -> list[V]:
    return [func(a, b) for a, b in zip(list1, list2)]

sums = zip_with(lambda a, b: a + b, [1, 2, 3], [10, 20, 30])
# sums: list[int] = [11, 22, 33]
```

# Bounded Type Variables

`bound=Animal` restricts `T` to `Animal` and its subclasses, which does two things simultaneously:

1. It prevents callers from passing unrelated types (no strings, no integers)
2. It unlocks the function body to safely call any method defined on `Animal`.

Without the bound, an unbounded `T` would accept any type but the body couldn't assume `.speak()` exists.

- With the bound, checker knows the method is always present regardless of whether the caller passes `Dog`, `Cat`, or a future `Parrot` subclass.

# Bounded Type Variables

```python
from typing import TypeVar

# Unbounded: T can be any type
T = TypeVar('T')

# Bounded: T must be subclass of specific type
class Animal:
    def speak(self) -> str:
        return "..."

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

class Cat(Animal):
    def speak(self) -> str:
        return "Meow!"

# T must be Animal or subclass
AnimalT = TypeVar('AnimalT', bound=Animal)

def make_speak(animal: AnimalT) -> str:
    return animal.speak()  # OK: Animal has speak()

make_speak(Dog())   # OK: Dog is Animal
make_speak(Cat())   # OK: Cat is Animal
make_speak("hello") # Error: str is not Animal!
```

# Constrained Type Variables

While `bound` allows any subclass of the specified type, constraints define an explicit allowlist.

- `TypeVar('Numeric', int, float)` means `T` can only be `int` or `float`, and nothing else, including subclasses.
- The difference from `Union[int, float]` in the return type is that a constrained TypeVar preserves which specific type was passed.
  - Calling `add(1, 2)` returns `int`
  - Calling `add(1.5, 2.5)` returns `float`
- Whereas a `Union` return would only say 'either int or float', losing the input-output relationship that callers need.

```
from typing import TypeVar

Numeric = TypeVar('Numeric', int, float)

def add(a: Numeric, b: Numeric) -> Numeric:
    return a + b

add(1, 2)       # OK: both int
add(1.5, 2.5)   # OK: both float
add("a", "b")   # Error: str not in constraints!
```

# Generic Classes

While generic functions apply a type parameter to a single function, `Generic[T]` extends this to an entire class.

- Every method that takes or returns the stored value shares the same `T`, enforcing consistency across the whole interface.
- When you write `Box[int]`, every occurrence of `T` in the class is resolved to `int`.
  - `get()` returns `int`, and `set()` only accepts `int`.
- The type checker then treats `Box[int]` and `Box[str]` as completely different types, just as `list[int]` and `list[str]` are, so passing a string to an int-box's `set()` method is caught statically rather than silently accepted and only discovered when the wrong type surfaces later.

# Generic Classes

```python
from typing import TypeVar, Generic

T = TypeVar('T')

class Box(Generic[T]):
    """A box that holds a value of type T"""

    def __init__(self, value: T) -> None:
        self._value = value

    def get(self) -> T:
        return self._value

    def set(self, value: T) -> None:
        self._value = value

# Usage with specific types
int_box: Box[int] = Box(42)
str_box: Box[str] = Box("hello")

value1: int = int_box.get()  # Type: int
value2: str = str_box.get()  # Type: str

int_box.set(100)     # OK
int_box.set("oops")  # Type error!
```

# Generic Stack - Definition

A stack is a last-in, first-out (LIFO) data structure where items are added to and removed from the same end.

- It's one of the use cases for a generic class because the logic is identical regardless of what you're storing.
- By parameterizing it as `Stack[T]`, every method that touches the stored items (`push`, `pop`, `peek`) is linked to the same type parameter, creating a coherent contract.
  - Once you create a `Stack[int]`, checker guarantees you can only push integers and that every pop returns an integer.
  - The internal storage `self._items: list[T]` inherits the same parameter, meaning the generic constraint flows through to the underlying Python list without any extra annotations.

# Generic Stack - Definition

```python
from typing import TypeVar, Generic

T = TypeVar('T')
class Stack(Generic[T]):
    def __init__(self) -> None:
        self._items: list[T] = []

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

    def pop(self) -> T:
        if not self._items:
            raise IndexError("Stack is empty")
        return self._items.pop()

    def peek(self) -> T:
        if not self._items:
            raise IndexError("Stack is empty")
        return self._items[-1]

    def is_empty(self) -> bool:
        return len(self._items) == 0
```

# Generic Stack - Usage

The benefit of a generic `Stack[T]` over a plain stack that stores `Any` or `object` is that every pop returns a known type rather than an unknown one.

- When you pop a value from `int_stack`, Python knows the result is `int`.
- With a non-generic stack storing `Any`, every pop returns `Any` and checker goes silent: you could accidentally assign the result to a `str` variable or call a string method on it, and the type checker would not warn you.
- Type inference also means you rarely need to write the type parameter explicitly
- If you push integers first, Python infers `Stack[int]` and will flag it if you accidentally push a string later.

```
# Using the Stack class
int_stack: Stack[int] = Stack()
int_stack.push(1)
int_stack.push(2)
print(int_stack.pop())  # 2, type: int

str_stack: Stack[str] = Stack()
str_stack.push("hello")
str_stack.push("world")
print(str_stack.pop())  # "world", type: str
```

# Multiple Generic Parameters

When a class represents a relationship between two independently varying types (a key-value pair, a first-second element, an input-output mapping) a single type variable can't capture the distinction between them because it would force both to be the same type.

- Two separate type parameters `K` and `V` make the relationship explicit
- `Pair[str, int]` and `Pair[int, str]` are different types to the checker, and `swap()` returning `"Pair[V, K]"` expresses that the two parameters flip rather than remaining in the same order.
- This level of type precision prevents entire classes of bugs where the first and second elements get confused, which is particularly valuable in data transformation pipelines.

# Multiple Generic Parameters

```python
from typing import TypeVar, Generic

K = TypeVar('K')
V = TypeVar('V')

class Pair(Generic[K, V]):
    """A pair of values with different types"""

    def __init__(self, first: K, second: V) -> None:
        self.first = first
        self.second = second

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

# Usage
pair: Pair[str, int] = Pair("age", 25)
print(pair.first)   # "age" (str)
print(pair.second)  # 25 (int)

swapped: Pair[int, str] = pair.swap()
```

# Multiple Generic Parameters - BiDict Example

A bidirectional dictionary maintains two synchronized internal maps

1. One from keys to values
2. One from values back to keys

This enables efficient lookup in either direction.

The two generic parameters `K` and `V` capture the structural symmetry between the two maps at the type level

- `_forward: dict[K, V]` and `_backward: dict[V, K]` express that the second map is the exact inverse of the first, with the types swapped.

# Multiple Generic Parameters - BiDict Example

```
class BiDict(Generic[K, V]):
    """Bidirectional dictionary - lookup by key OR value"""

    def __init__(self) -> None:
        self._forward: dict[K, V] = {}
        self._backward: dict[V, K] = {}

    def set(self, key: K, value: V) -> None:
        self._forward[key] = value
        self._backward[value] = key

    def get_by_key(self, key: K) -> V:
        return self._forward[key]

    def get_by_value(self, value: V) -> K:
        return self._backward[value]
```

# Generic with Inheritance

When subclassing a generic class, you face a design choice:

- Specialize the type parameter to a concrete type (making the subclass non-generic but focused on specific behavior),
- Or forward the TypeVar (keeping the subclass generic so it works for any type).
- `IntContainer(Container[int])` takes the first path and it commits to integers and gains the ability to add integer-specific methods like `increment()` that wouldn't make sense for strings or other types.
- `LoggedContainer(Container[T])` takes the second path and it adds logging behavior that works regardless of the stored type, staying fully generic and usable wherever the original `Container[T]` is expected.

# Generic with Inheritance

```python
from typing import TypeVar, Generic

T = TypeVar('T')

class Container(Generic[T]):
    def __init__(self, value: T) -> None:
        self._value = value

    def get(self) -> T:
        return self._value

# Inherit and SPECIALIZE to concrete type
class IntContainer(Container[int]):
    def increment(self) -> None:
        self._value += 1

# Inherit and REMAIN generic
class LoggedContainer(Container[T]):
    def get(self) -> T:
        print(f"Accessing value: {self._value}")
        return super().get()

int_cont = IntContainer(5)
int_cont.increment()  # Works - int specific

logged: LoggedContainer[str] = LoggedContainer("hello")
logged.get()  # Logs then returns
```

# Advanced Typing

# Callable Type

In Python, functions are first-class objects meaning they can be stored in variables, passed as arguments to other functions, and returned as values.

- `Callable` is the type annotation that expresses this.

The syntax `Callable[[ArgType1, ArgType2], ReturnType]` reads as '*a function that accepts ArgType1 and ArgType2 and returns ReturnType*', with argument types always written as a list even for a single argument.

- When you don't care about the argument types (for example, a callback that might have any signature), `Callable[..., ReturnType]` uses an ellipsis as a wildcard meaning '*any arguments are acceptable*'.
- Without `Callable`, passing the wrong kind of function to a higher-order function would only fail at runtime when the function is actually called with incompatible arguments.

# Callable Type

```python
from typing import Callable

# Function that takes int and returns str
def process(func: Callable[[int], str]) -> str:
    return func(42)

def int_to_str(n: int) -> str:
    return str(n)

result = process(int_to_str)  # "42"

# Multiple arguments
def combine(func: Callable[[int, int], int], a: int, b: int) -> int:
    return func(a, b)

combine(lambda x, y: x + y, 1, 2)  # 3

# Variable arguments (use ...)
def execute(func: Callable[..., int]) -> int:
    return func(1, 2, 3, key="value")
```

# Callable in Practice - Retry Pattern

The `retry` function illustrates two complementary uses of `Callable` together:

1. `Callable[[], T]` types a zero-argument block.
2. `Callable[[Exception], None]` types the optional error handler that receives the exception and returns nothing.

Using a `TypeVar` for the return type `T` makes the entire retry wrapper generic.

- If the wrapped function returns `int`, then `retry(func)` returns `int | None`, preserving type information all the way through the wrapper.

# Callable in Practice - Retry Pattern

```python
from typing import Callable, TypeVar

T = TypeVar('T')

def retry(
    func: Callable[[], T],
    max_attempts: int = 3,
    on_failure: Callable[[Exception], None] | None = None
) -> T | None:
    for attempt in range(max_attempts):
        try:
            return func()
        except Exception as e:
            if on_failure:
                on_failure(e)
            if attempt == max_attempts - 1:
                raise
    return None
```

# Callable in Practice - Composition

`Callable[[T], U]` gives `map_custom` a solid contract.

- It accepts any function from `T` to `U` and a list of `T`, producing a list of `U`, both `T` and `U` are inferred separately at each call site.

When we write `map_custom(str, [1, 2, 3])`, the checker infers `T=int` from the list and `U=str` from `str`'s return annotation, marking the result as `list[str]` automatically.

- This is the properly typed version of Python's built-in `map()`, which returns `Iterable[Any]` because it can't express the relationship between the function's type and the output typ.

```
from typing import Callable, TypeVar

T = TypeVar('T')
U = TypeVar('U')

def map_custom(func: Callable[[T], U], items: list[T]) -> list[U]:
    """Apply func to each item, preserving type information"""
    return [func(item) for item in items]

# Type checker infers return types from func signature
strings = map_custom(str, [1, 2, 3])        # list[str]
lengths = map_custom(len, ["hi", "hello"])   # list[int]
doubled = map_custom(lambda x: x * 2, [1, 2, 3])  # list[int]
```

# ParamSpec for Decorators

When we write a decorator using the naive approach, typing the wrapper as `Callable[..., R]`, we tell Python '*this returns R and accepts anything*', which means the decorated function loses all parameter checking and IDE autocomplete disappears for callers.

- `ParamSpec('P')` solves this by capturing the complete parameter specification of the wrapped function as a type variable, including all argument names, types, and defaults.
- The wrapper then uses `*args: P.args` and `**kwargs: P.kwargs` to forward arguments without changing their types, so `@logged` on `add(a: int, b: int) -> int` preserves the full signature: Python still enforces that callers pass two integers even though the function being executed is the wrapper.

# ParamSpec for Decorators

```python
from typing import Callable, ParamSpec, TypeVar

P = ParamSpec('P')  # Captures parameter types
R = TypeVar('R')    # Return type

def logged(func: Callable[P, R]) -> Callable[P, R]:
    def wrapper(*args: P.args, **kwargs: P.kwargs) -> R:
        print(f"Calling {func.__name__}")
        result = func(*args, **kwargs)
        print(f"Returned {result}")
        return result
    return wrapper

@logged
def add(a: int, b: int) -> int:
    return a + b

result: int = add(1, 2)  # OK
add("x", "y")  # Type error: expected int, got str
```

# TypedDict

Plain `dict[str, Any]` is the least informative type we can attach to a dictionary.

- It tells Ptyhon nothing about which keys exist or what types their values are, so accessing `user['name']` or `user['age']` can't be verified.

`TypedDict` gives dictionaries a schema: we declare each required key and its type using class-body syntax, and mypy then validates every key access, catches typos in key names, and flags missing fields when a TypedDict is constructed.

- This is particularly valuable for data from external sources (JSON API responses, database rows, configuration files) where we know the structure but would otherwise have to cast everything from `Any`.

Unlike dataclasses or attrs, a `TypedDict` instance is still a plain Python dictionary at runtime with no wrapper class, no `__init__`, and no overhead and the schema exists only for the static checker.

# TypedDict

```python
from typing import TypedDict

class UserDict(TypedDict):
    name: str
    age: int
    email: str

def process_user(user: UserDict) -> str:
    return f"{user['name']} ({user['age']})"

user: UserDict = {
    "name": "Alice",
    "age": 30,
    "email": "alice@example.com"
}

process_user(user)  # OK
process_user({"name": "Bob"})  # Error: missing keys!
```

# TypedDict - Optional Keys

By default, all fields in a `TypedDict` are required, Python will flag any construction that omits a key.

- `total=False` changes this and all fields become optional, meaning a dict with any subset of the declared keys is valid.

For cases where we need a mix (e.g., some keys always present, others optional) `Required` and `NotRequired` let us annotate each field individually within a `total=False` TypedDict, giving use fine-grained control without splitting the definition into two classes.

- This is the idiomatic pattern for PATCH-style API payloads where a caller sends only the fields they want to update, or for configuration objects where sensible defaults exist for most but not all fields.

```
from typing import TypedDict, Required, NotRequired

class PartialUser(TypedDict, total=False):
    name: Required[str]   # Always required
    age: NotRequired[int] # Explicitly optional
    email: str            # Optional (total=False applies)

partial: PartialUser = {"name": "Charlie"}  # OK!
full: PartialUser = {"name": "Alice", "age": 30, "email": "a@b.com"}
```

# Type Guards

The type checker automatically narrows types inside:

- `if isinstance(x, str): ...` blocks once inside the if body, Python knows `x` is a `str`.
- But this narrowing only works for `isinstance` and `is None` checks, not for custom validation functions.

`TypeGuard[T]` extends type narrowing to arbitrary predicate functions: when a function annotated with `-> TypeGuard[list[str]]` returns `True`, Python treats that as proof that the argument is of type `list[str]` in any code that follows.

- Without `TypeGuard`, even if we manually check `all(isinstance(x, str) for x in items)`, Python still sees `items` as `list[object]` inside the if block and rejects `.upper()` calls because it doesn't know our check implies the type.

# Type Guards

```python
from typing import TypeGuard

def is_string_list(val: list[object]) -> TypeGuard[list[str]]:
    """Check if all items in list are strings"""
    return all(isinstance(item, str) for item in val)

def process(items: list[object]) -> None:
    if is_string_list(items):
        # Type checker now knows items is list[str]!
        for item in items:
            print(item.upper())  # OK: item is str
    else:
        print("Not all strings")

# Without TypeGuard
def old_process(items: list[object]) -> None:
    if all(isinstance(item, str) for item in items):
        for item in items:
            print(item.upper())  # Error: item is still object!

# TypeGuard tells type checker about the type narrowing
```

# Type Checking with mypy

mypy is the standard static type checker for Python.

- It reads source files, follows imports, checks annotations against usages, and reports errors before any code executes.
- Unlike linters that check style or potential bugs heuristically, mypy does full type inference: it understands the return types of standard library functions, follows generic type parameters through function calls, and resolves overloaded signatures.

The `strict` flag in `pyproject.toml` activates the full ruleset:

- Requiring annotations on every function, disallowing implicit `Any`, and warning when we return `Any` from an annotated function which is the recommended setting for new projects that want maximum safety.

```
pip install mypy

# Check a single file or entire project
mypy script.py
mypy src/
```

```
# pyproject.toml
[tool.mypy]
python_version = "3.11"
strict = true
warn_return_any = true
warn_unused_configs = true
```

# mypy in Action

When mypy finds a type error, it reports the exact file, line number, and a description of the mismatch

- `Argument 1 to "greet" has incompatible type "int"; expected "str"`

This happens entirely offline, before the code runs.

```
# example.py
def greet(name: str) -> str:
    return "Hello, " + name

greet(42)  # Type error

# mypy output:
# example.py:4: error: Argument 1 to "greet" has
#   incompatible type "int"; expected "str"

# Works with class hierarchies too
def process(items: list[str]) -> None:
    for item in items:
        print(item.upper())

process([1, 2, 3])  # Error: list[int] is not list[str]
```

# mypy Strictness Levels

mypy is designed for gradual adoption.

- A codebase with no annotations at all is still valid mypy input, it just has nothing to check.
- As we add annotations incrementally, mypy checks more and more of the code, letting us introduce type safety file by file or module by module without blocking the rest of the team.

The individual flags in `pyproject.toml` let us tune exactly how strict the checker is:

- `disallow_untyped_defs` requires annotations on every function definition
- `warn_return_any` flags functions that return `Any` from typed code
- `check_untyped_defs` analyzes function bodies even when no annotations are present.

This granularity means we can start with a single flag, fix the warnings it surfaces, then tighten further rather than enabling `strict` all at once and facing thousands of errors.

# mypy Strictness Levels

```python
# Without annotations — mypy has nothing to check
def add(a, b):
    return a + b
```

Individual rules in `pyproject.toml`:

```toml
[tool.mypy]
disallow_untyped_defs = true   # Require annotations on all functions
disallow_any_generics = true   # No bare List, Dict without type params
warn_return_any = true         # Warn when returning Any
warn_unused_ignores = true     # Warn on stale # type: ignore
check_untyped_defs = true      # Still check bodies without annotations
```

# Handling Type Checker Issues

Even with a well-typed codebase, mypy occasionally gets things wrong usually when interfacing with third-party libraries that lack type stubs, when using dynamic features mypy can't follow, or when the type system isn't expressive enough to capture a particular invariant.

- `# type: ignore` is the escape tool for those situations: it silences the error on that specific line, and adding a reason code like `[no-untyped-call]` documents why the suppression is legitimate so future maintainers don't remove it blindly.
- `cast(T, value)` is different. It doesn't change runtime behavior at all, it just tells mypy '*trust me, this value is of type T*'.
  - Useful after `json.loads` or other operations that return `Any`.
- `reveal_type(x)` is a debugging tool we insert temporarily: mypy reports the inferred type of `x` at that line, helping us understand why downstream errors are occurring.

# Handling Type Checker Issues

```
# Ignore specific line
result = some_untyped_function()  # type: ignore

# Ignore with reason (better!)
result = legacy_function()  # type: ignore[no-untyped-call]

# Cast when you know better than type checker
from typing import cast

data = json.loads(text)  # Returns Any
user = cast(UserDict, data)  # Tell checker it's UserDict

# assert for type narrowing
def process(value: str | int) -> str:
    assert isinstance(value, str)  # Narrows to str
    return value.upper()  # OK now

# reveal_type for debugging (removed at runtime)
x = [1, 2, 3]
reveal_type(x)  # mypy shows: builtins.list[builtins.int]
```

# Overload for Multiple Signatures

When a function's return type depends on which input type it receives, annotating the return as `str | int | list[str]` loses all precision because callers know the function returns one of those three things, but the type checker can't tell which one based on what was passed in.

- `@overload` solves this by letting us declare multiple separate signatures, each pairing a specific input type with its corresponding output type.
    - The type checker uses whichever overload signature matches the caller's argument, giving callers the exact return type rather than a union.
- The actual implementation body must accept and return a union that covers all cases, but that body is used only for the runtime implementation and callers see only the cleaner overloaded signatures.

# Overload for Multiple Signatures

```python
from typing import overload

@overload
def process(value: str) -> str: ...

@overload
def process(value: int) -> int: ...

@overload
def process(value: list[str]) -> list[str]: ...

def process(value: str | int | list[str]) -> str | int | list[str]:
    if isinstance(value, str):
        return value.upper()
    elif isinstance(value, int):
        return value * 2
    else:
        return [v.upper() for v in value]

# Type checker knows specific return types!
s: str = process("hello")      # Returns str
n: int = process(5)            # Returns int
```

# Self Type (Python 3.11+)

Fluent interfaces and builder patterns rely on method chaining each method returns `self` so the next call can be chained immediately.

- The problem is that annotating the return as `-> "Builder"` is too specific:
  - `PersonBuilder` inherits from `Builder`, calling `.with_name('Alice')` is seen as returning a `Builder`, not a `PersonBuilder`, so mypy rejects the subsequent `.with_email()` call as a type error because `Builder` doesn't have that method.

`Self` (from `typing` in Python 3.11+) solves this issue. It's a special type variable that always resolves to the actual class of the instance at the call site:

- `PersonBuilder().with_name('Alice')` returns `PersonBuilder`, and chaining `.with_email()` is valid.
- This makes `Self` the correct annotation for any method that returns `self` and is intended to be used in inheritance hierarchies.

# Self Type (Python 3.11+)

```python
from typing import Self

class Builder:
    def __init__(self) -> None:
        self._name: str = ""
        self._age: int = 0

    def with_name(self, name: str) -> Self:
        self._name = name
        return self

    def with_age(self, age: int) -> Self:
        self._age = age
        return self

class PersonBuilder(Builder):
    def with_email(self, email: str) -> Self:
        self._email = email
        return self

# Method chaining works correctly with inheritance
person = (PersonBuilder()
    .with_name("Alice")    # Returns PersonBuilder
    .with_email("a@b.com") # Works!
)
```

# Thank You!

## Contact Information

- **Email:** ekrem.cetinkaya@yildiz.edu.tr
- **Office Hours:** Wednesday 13:30-15:30 - Room C-120
- **Book a slot before coming:** Booking Link
- **Course Repository:** GitHub

## Next Week

- **Week 13:** Concurrent Programming: Threads and Synchronization