Lecture 02: Types_And_Pattern_Matching

Vincenzo Ciancia

April 04, 2025

Section 1: Python’s Type System

What are Type Annotations?

Python's type system allows developers to add optional type hints to variables, function parameters, and return values. These annotations help catch errors early, improve code documentation, and enhance IDE support without changing the runtime behavior of the code.

```python
def greet(name: str) -> str:
    return f"Hello, {name}!"
```

Type annotations are part of the Python Enhancement Proposal (PEP) 484 and have been continuously improved in subsequent PEPs. They provide a way to make Python code more robust through static type checking, although Python remains dynamically typed at runtime.

Basic Types in Python

Python provides several built-in types for annotations:

- **Primitive types**: `int`, `float`, `bool`, `str`
- **Collection types**: `list`, `tuple`, `dict`, `set`

Example from `types_and_matching.py`:

```python
my_tuple: tuple[int, str, float] = (1, "hello", 1.0)

my_list: list[int] = [1, 2, 3, 4, 5]
```

These annotations tell the type checker that `my_tuple` is a 3-element tuple containing an integer, a string, and a float, while `my_list` is a list containing only integers.

Generic Types

Generic types allow you to create reusable, type-safe components. In Python 3.12+, the syntax for generic classes uses square brackets:

```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 | None:
        return self.items.pop() if self.items else None
```

In this example from our code: - `T` is a type parameter representing any type - `Stack[T]` is a generic class that can be specialized for specific types - `list[T]` indicates a list containing elements of type `T`

To use this generic class:

```python
stack_1 = Stack[int]()  # A stack of integers
```

Union Types and Optional Values

Union types allow a variable to have multiple possible types, expressed using the `|` operator (introduced in Python 3.10):

```python
def process_data(data: int | str) -> None:
    # Can handle both integers and strings
    pass
```

From our example code:

```python
def pop(self) -> T | None:  # Use | for union types
    return self.items.pop() if self.items else None
```

This indicates that the `pop` method returns either a value of type `T` or `None` if the stack is empty.

Type Aliases

Type aliases help simplify complex type annotations:

```
type JsonData = dict[str, int | str | list | dict]

type A = int | dict[str, A]  # Recursive type
```

In `types_and_matching.py`, we use type aliases for recursive types:

```
type MyBaseList[T] = None | MyList[T]

type Expr = int | Sum
```

These aliases make the code more readable and allow for recursive type definitions.

Literal Types

Literal types restrict values to specific constants:

```
y: Literal["hello", "world"] = "hello"  # Valid

z: Literal[1, 2, 3] = 9  # Type error
```

From our example code:

```
@dataclass

class Human:

    name: str

    drivingLicense: Literal[True, False]
```

This constrains the `drivingLicense` attribute to be either `True` or `False` only.

Section 2: Structural Pattern Matching

Introduction to Pattern Matching

Introduced in Python 3.10, the `match` statement provides powerful pattern matching capabilities, similar to switch statements in other languages but with more expressive power:

```python
def describe(value):
    match value:
        case 0:
            return "Zero"
        case int(x) if x > 0:
            return "Positive integer"
        case str():
            return "String"
        case _:
            return "Something else"
```

Pattern matching allows for more concise and readable code, especially when dealing with complex data structures and multiple conditions.

Basic Patterns

Basic patterns match against simple values and types:

```
match x:
    case 0:
        print("Zero")
    case int():
        print("Integer")
    case str():
        print("String")
    case _:  # Wildcard pattern
        print("Something else")
```

The _ pattern is a wildcard that matches anything and is often used as a catch-all case.

Sequence Patterns

Sequence patterns match against sequence types like lists and tuples:

```
def process_lst(lst: list[int]) -> None:
    match lst:
        case []:
            print("List: empty")
        case [head, *tail]:
            print(f"List: head {head}, tail {tail}")
```

This example shows destructuring a list into its head (first element) and tail (remaining elements), demonstrating how pattern matching facilitates recursive list processing.

Class Patterns and Attribute Matching

Pattern matching works particularly well with dataclasses:

```python
def greet(person: Person | str) -> None:
    match person:
        case Person(name="Alice", age=25):
            print("Hello Alice!")
        case Person(name=x, age=y):
            print(f"Hello {x} of age {y}!")
        case str():
            print(f"Hello {person}!")
```

This example shows matching against specific attribute values and binding attributes to variables.

Complex Pattern Matching Examples

Recursive Pattern Matching

Pattern matching excels at handling recursive data structures:

```
def sum_list_2(lst: MyBaseList[int]) -> int:
    match lst:
        case None:
            return 0
        case MyList(head=x, tail=y):
            return x + sum_list_2(y)
```

This function processes a custom linked list structure using pattern matching to handle the base case (None) and recursive case elegantly.

Parsing Expressions

Pattern matching can implement simple interpreters:

```python
def eval_expr(expr: Expr) -> int:
    match expr:
        case int(x):
            return x
        case Sum(left=x, right=y):
            return eval_expr(x) + eval_expr(y)
```

This code evaluates a simple arithmetic expression tree, showing how pattern matching simplifies traversal of complex structures.

Section 3: Combining Types and Pattern Matching

Algebraic Data Types in Python

Python can implement algebraic data types (ADTs) using classes, dataclasses, and union types:

Sum Types (Tagged Unions)

Sum types represent values that could be one of several alternatives:

```
@dataclass
class Human:
    name: str
    drivingLicense: Literal[True, False]


@dataclass
class Dog:
    name: str
    kind: DogKind
    colour: Literal["brown", "black", "white"]
```

Section 4: Applications and Best Practices

When to Use Type Annotations

Type annotations are particularly valuable in: - Large codebases with multiple developers - APIs and libraries that will be used by others - Performance-critical code where type-specific optimizations matter - Complex data processing pipelines

Domain-specific languages, interpreters, ADTs!!

When to Use Pattern Matching

Pattern matching excels at: - Processing complex recursive data structures - Implementing interpreters and compilers - Handling case-based logic with destructuring - Processing structured data like JSON or ASTs

Best Practices for Type Annotations

1. Be consistent with type annotations across your codebase
2. Use type aliases for complex or repetitive type expressions
3. Leverage tools like mypy, pyright, or pylance for static type checking
4. Balance between type precision and code readability

Exercises

- Consider the expression evaluator written in Lecture 01. Turn it into a parser that converts a string of the form "x op y op z ..." separated by spaces into an AST in the sense of Lecture 02. Concatenate the new parser and the evaluation function that takes AST as input to define a mini-interpreter.
- Add the "$==$" boolean operator to the AST and the evaluator.

Additional Resources

- PEP 484 – Type Hints
- PEP 585 – Type Hinting Generics In Standard Collections
- PEP 604 – Allow writing union types as X | Y
- PEP 634 – Structural Pattern Matching: Specification
- PEP 636 – Structural Pattern Matching: Tutorial
- Mypy Type Checker Documentation
- Real Python: Python Type Checking
- Real Python: Structural Pattern Matching in Python