Basic Type Hint Syntax

Annotating Functions and Variables

Effective Programming Practices for Economists

Function Annotations

```
def function_name(arg1: Type1, arg2: Type2) -> ReturnType:
    ...
```

Example:

```
def calculate_mean(values: list, n: int) -> float:
    return sum(values) / n
```

- Arguments: `arg: Type`
- Return type: `-> Type`

Built-in Types

```python
def example(
    count: int,           # Integers
    price: float,         # Floating point
    name: str,            # Strings
    is_valid: bool,       # Booleans
    data: bytes,          # Binary data
) -> None:                # Function returns nothing
    pass
```

Use lowercase for built-in types

`int`, `float`, `str`, `bool`, `bytes`, `None`

Variable Annotations

```
count: int = 0
name: str = "Alice"
prices: list = [1.99, 2.49, 3.99]

# Annotation without assignment (declares type)
result: float
```

When to annotate variables?

- When the type isn't obvious from the assignment
- When declaring before assignment
- Usually not necessary (IDE can infer)

Union Types

```python
# Accept multiple types
def process_id(id_value: int | str) -> str:
    return str(id_value)

process_id(3)        # OK
process_id("abc")    # OK
process_id(3.14)     # Type error!
```

Use `|` to combine types

- `int | str` means "either int or str"
- Can chain: `int | str | float`
- Can use for optional values: `str | None`

Default Arguments

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

- Type hint comes before the default value
- Pattern: `name: Type = default`

Multiple Return Values

```python
def get_stats(values: list[float]) -> tuple[float, float]:
    """Return mean and standard deviation."""
    mean = sum(values) / len(values)
    variance = sum((x - mean) ** 2 for x in values) / len(values)
    return mean, variance ** 0.5

# Usage
mean, std = get_stats([1, 2, 3, 4, 5])
```

Use `tuple[Type, ...]` an unknown number of same-type elements

*args and **kwargs

```python
def log_values(*args: float, **kwargs: str) -> None:
    """Log numeric values with string metadata."""
    for value in args:
        print(f"Value: {value}")
    for key, value in kwargs.items():
        print(f"{key}: {value}")

log_values(1.0, 2.0, 3.0, source="sensor", unit="celsius")
```

Annotate the element type, not the container

- `*args: float` means each arg is a float
- `**kwargs: str` means each value is a str

Type Aliases

```python
type UserId = int
type Coordinates = tuple[float, float]
type Matrix = list[list[float]]

def get_user_location(user_id: UserId) -> Coordinates:
    ...

def transpose(matrix: Matrix) -> Matrix:
    ...
```

Benefits:

- Self-documenting code
- Easy to change type in one place
- More readable signatures

The Any Type

```python
from typing import Any

def log_anything(value: Any) -> None:
    """Accept any type - use sparingly!"""
    print(value)
```

`Any` disables type checking

- Use when interacting with untyped code
- Use during migration to typed code
- **Avoid** in new code when possible

Common Mistake

Wrong: Using `list` without element type

```
def bad(items: list) -> int:  # What's in the list?
    return len(items)
```

Right: Specify element type

```
def good(items: list[int]) -> int:
    return len(items)
```

Quick Reference

Type	Example
Integer	`x: int = 3`
Float	`x: float = 3.14`
String	`x: str = "hello"`
Boolean	`x: bool = True`
None	`x: None = None`
Optional	`x: str | None = None`
Union	`x: int | str`

Summary

Basic syntax:

- `arg: Type` for parameters
- `-> Type` for return values
- `var: Type = value` for variables

Key types:

- Built-ins: `int`, `float`, `str`, `bool`, `None`
- Optional: `Type | None`
- Union: `Type1 | Type2`