# Good Coding Practices

**Definition**
Quality code is maintainable and easy to follow.

## Using Type Checkers and Linters

### Python Type Hints

Python support type hints, see example below.

```
pi: float = 3.14159
is_student: bool = True

def my_func(name: str, age: int, friends: list[str]) -> bool:
  # do stuff
```

Type hints can be used with a **type checker** to look for bugs, but are not checked at runtime by Python. Note that the `any` type disables the type checker for that variable. (danger!)

**mypy** is a type checker for python.

```
mypy --disallow-untyped-defs <file>
```

### Generic Types

Python supports the following generic collections:

- `list[T]`
- `set[T]`
- `dict[K, V]`
- `tuple[A, B, C, ...]`
- `Callable[[arg1, arg2, ...], return_type]`

### Type Aliases

Some examples below.

```
IntList = list[int]
WordDefinitions = dict[str, str]
```

Type aliases are **not** classes, the code below will raise no errors from the type checker.

```
IntList1 = list[int]
IntList2 = list[int]

def my_func(list: IntList1):
  # do stuff

my_func(IntList2()) # no errors from type checker
```

**Definition**
`Optional[X] = X | None`

### Python Linting

A linter checks for coding styles and likely errors. **pylint** is a linter for python.

**Definitions**

- A **unit** is the smallest testable piece of code, e.g. a single function/method.
- A **unit test** automatically verifies that a specific **unit** works as intended.

### Using `pytest`

**Automatic discovery** automatically looks for test functions.

| What | Format |
|---|---|
| File | `test_*.py` or `*_test.py` |
| Function | `test_*` |

### Raising Errors

```
# Using assertions
assert boolean_value message

# Raising a particular error
raise ValueError("Message")
```

A unit test can expect an error, this test fails if no `ValueError` is raised.

```
def test_div_by_zero_error():
  with pytest.raises(ValueError):
    divide(1, 0)
```

### Test Driven Development

TDD focuses on the feature to implemented rather than the code needed to implement it.

1. Write a failing test defining the desired functionality.
2. Write the minimum amount of code required to pass the test.
3. Go to step 1.