IRENA GRGIC

Clean up your code

GeekOut!

code.talks

Introduction

Irena Grgic, Lead DevOps Engineer at Carl Zeiss

Follow me on:

irac.grgic @Medium

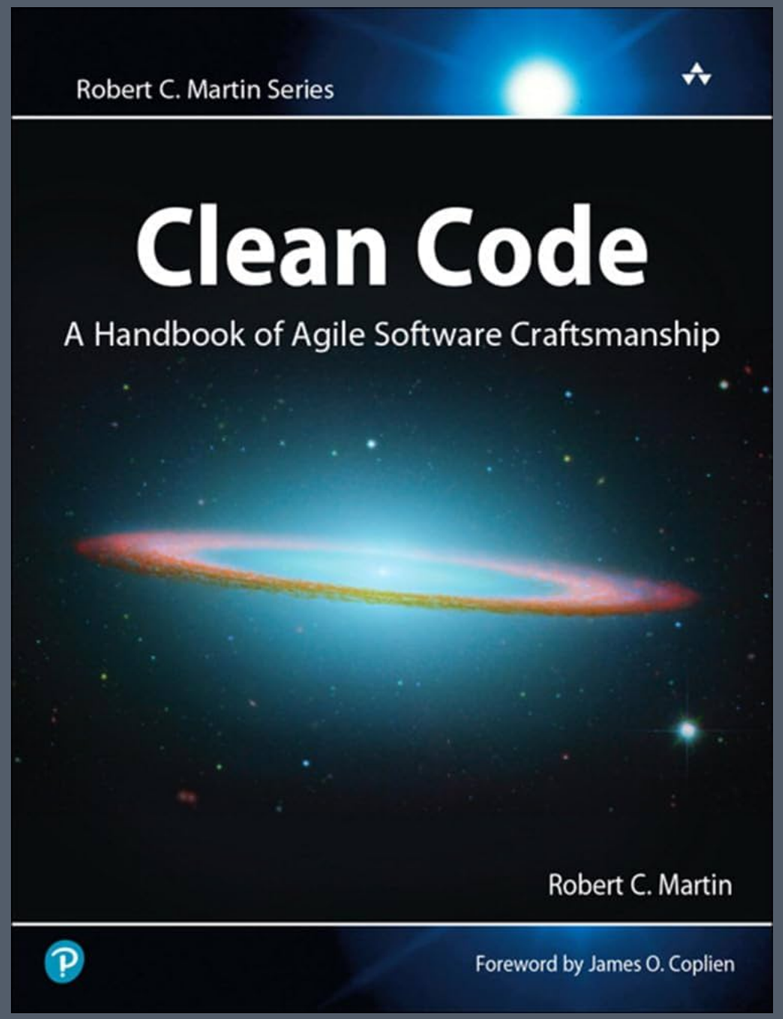
LinkedIn

pythonmonty @GitHub

Clone the repository

https://github.com/pythonmonty/clean-code-in-python

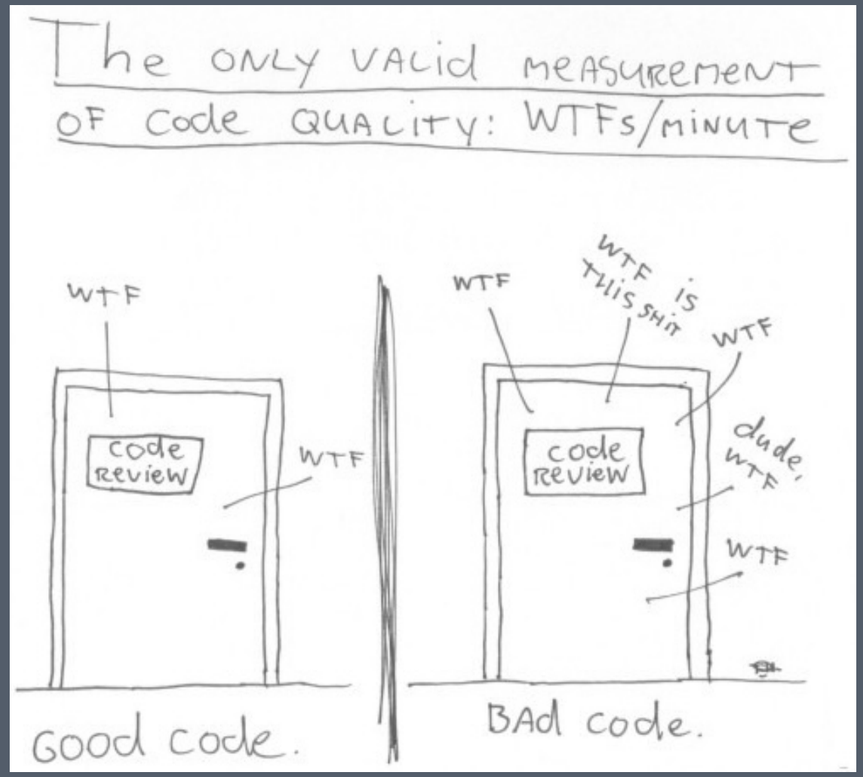
Clean Code, by Robert C. Martin (Uncle Bob)



What is bad code?

Bad code is like a building with broken windows.

The measurement of code quality[1]



[1] Image source

What is clean code?

According to Bjarne Stroustrup, inventor of C++

I like my code to be elegant[2] and efficient.

[2] *Elegant: pleasingly ingenious and simple (Oxford Dictionary).*

Clean code does one thing well.

Environments

Use a tool to manage your Python versions

→ pyenv
→ conda or miniconda
→ asdf

Set version ranges for packages

Bad:

```
# requirements.txt
pandas ⩾ 1.5.0
```

Set version ranges for packages

Also bad:

```
# requirements.txt
scikit-learn
onnx
```

Set version ranges for packages

Better?

```
# requirements.txt
pandas==1.5.0
```

Renovate is a good tool that automatically upgrades packages for you.

Don't use `pip` and `requirements.txt` for complex scenarios

```
.
├── requirements.txt
├── requirements_dev.txt
├── requirements_freeze.txt
├── requirements_mac.txt
├── requirements_test.txt
├── requirements_windows.txt
```

Use a dependency manager that enables:

- → A separation of dependency groups (e.g. `dev`, `test`).
- → Integration with `pyproject.toml`.
- → Hashed and locked package versions.
- → Different versions for different platforms.
- → Build and publishing with CI/CD integration.

Modern dependency managers that tick off all the boxes

→ Poetry

→ Hatch

→ PDM (Python Development Master)

Poetry

A separation of dependency groups in `pyproject.toml`

```
[tool.poetry.dependencies]
python = "^3.9"
pandas = "^2.2.2"

[tool.poetry.group.dev.dependencies]
ruff = "^0.5.7"
pre-commit = "^3.8.0"
mypy = "^1.11.1"

[tool.poetry.group.test.dependencies]
pytest = "^8.3.2"
```

Poetry

Hashed and locked package versions with `poetry.lock`

```
[[package]]
name = "pandas"
version = "2.2.2"
description = "Powerful data structures for data analysis, time series, and statistics"
optional = false
python-versions = "≥3.9"
files = [
    {file = "pandas-2.2.2-cp310-cp310-macosx_10_9_x86_64.whl", hash = "sha256:90c6fca2acf139569e74e8781709dccb6fe25940488755716d1d354d6bc58bce"},
    {file = "pandas-2.2.2-cp310-cp310-macosx_11_0_arm64.whl", hash = "sha256:c7adfc142dac335d8c1e0dcbd37eb8617eac386596eb9e1a1b77791cf2498238"},
    ...
```

Poetry

Different versions for different platforms

```
[tool.poetry.dependencies]
python = "^3.10"
torch = [
    { markers = "sys_platform ≠ 'linux'", version = "==2.1.2", source = "PyPI" },
    { markers = "sys_platform == 'linux'", version = "==2.1.2+cu121", source = "torchcuda" }
]

[[tool.poetry.source]]
name = "torchcuda"
url = "https://download.pytorch.org/whl/cu121"
priority = "explicit"
```

Exercise [3 min]

→ Clone the repository (if not already): https://github.com/pythonmonty/clean-code-in-python.

→ Check out the `README.md` of the repository and follow the *Installation* guide.

Naming

Use the PEP 8 style guide

- → Variables, functions, methods, module, package: `snake_case`
- → Classes: `CamelCase`
- → Constants: `SCREAMING_SNAKE_CASE`

Use intention-revealing names

→ Why does it exist?

→ What does it do?

→ How is it used?

Use intention-revealing names

Instead of this

```
rt = 5  # remaining time in seconds
```

do this

```
remaining_time_in_seconds = 5
```

Use searchable names

Do you really want to search for a variable called `t`?

```
from datetime import datetime



t = datetime.now().time()
```

Use searchable names

Better?

```
from datetime import datetime



current_time = datetime.now().time()
```

Use nouns for class names

Avoid names like `Processor`, `Reader`, `Manager`, `Data`, `Info`.

```
class Data:
    def __init__(self, data):
        self.data = data
```

Use verbs for function / method names

```
def train_classification_model():
    ...
```

```
def validate_project_name_regex_pattern():
    ...
```

Don't mix case styles

```
class ConfusionMatrix:
    ...

def generate_ConfusionMatrix():
    ...
```

Instead, pick a case style and stick with it:

```
def generate_confusion_matrix() → ConfusionMatrix:
    ...
```

Pick one word for an abstract concept and stick with it

Same abstract concept, different naming:

```
def generate_confusion_matrix():
    ...

def create_roc_curve():
    ...
```

Pick one word for an abstract concept and stick with it

Better:

```
def generate_confusion_matrix():
    ...

def generate_roc_curve():
    ...
```

Don't add unnecessary context

Instead of this

```
customers_list = ["Simone Biles", "Rebeca Andrade"]
```

do this

```
customers: list[str] = ["Simone Biles", "Rebeca Andrade"]
```

Don't add unnecessary context

Instead of this

```
class Customer:
    customer_name: str
    customer_surname: str
    customer_address: str
```

Don't add unnecessary context

do this:

```
class Customer:
    name: str
    surname: str
    address: str
```

Use meaningful and pronounceable names

```python
from torchvision.models.resnet import ResNet


def resnet50(
        *,
        weights: Optional[ResNet50_Weights] = None,
        progress: bool = True,
        **kwargs: Any,
) -> ResNet
```

Use meaningful and pronounceable names

Better?

```
def load_pre_trained_resnet_50_layers_model(
        *,
        weights: Optional[ResNet50_Weights] = None,
        progress: bool = True,
        **kwargs: Any,
) → ResNet
```

Use meaningful and pronounceable names

Another `torchvision` example:

```
def resnext101_32×8d(
    *,
    weights: Optional[ResNeXt101_32X8D_Weights] = None,
    progress: bool = True,
    **kwargs: Any,
) → ResNet:
```

Functions

Functions should be small

→ If you think that your function is small, make it even smaller.

→ If you have to scroll to read the function, it is too big.

Functions should do only one thing

→ they should do it well,

→ they should do it only.

Functions should do only one thing - Bad example

```
import datetime



class Employee:
    name: str
    surname: str
    type: str
```

Functions should do only one thing - Bad example

```
def calculate_salary(employee: Employee, date: datetime.date) → None:
    if employee.type == "comissioned":
        salary = calculate_comissioned_salary(employee)
    elif employee.type == "full-time":
        salary = calculate_full_time_salary(employee)
    else:
        raise UnsupportedEmployeeTypeError(f"Employee type {employee.type} "
                                           f"is not supported")
    if is_payday(employee, date):
        deliver_salary(employee, salary)
```

Functions should do only one thing - Bad example

```
def calculate_salary(employee: Employee, date: datetime.date) -> None:
    if employee.type == "comissioned":
        salary = calculate_comissioned_salary(employee)
    elif employee.type == "full-time":
        salary = calculate_full_time_salary(employee)
    else:
        raise UnsupportedEmployeeTypeError(f"Employee type {employee.type} "
                                           f"is not supported")
    if is_payday(employee, date):
        deliver_salary(employee, salary)
```

Functions should do only one thing - Bad example

```
def calculate_salary(employee: Employee, date: datetime.date) → None:
    if employee.type == "comissioned":
        salary = calculate_comissioned_salary(employee)
    elif employee.type == "full-time":
        salary = calculate_full_time_salary(employee)
    else:
        raise UnsupportedEmployeeTypeError(f"Employee type {employee.type} "
                                           f"is not supported")
    if is_payday(employee, date):
        deliver_salary(employee, salary)
```

One level of abstraction per function

→ Statements within a function should be on the same level of abstraction.

→ When reading the code, every function should be followed by those at the next level of abstraction.

One level of abstraction per function - Bad example

```python
def process_user_data(user: User) -> str:
    validate_user_data(user)

    full_name = (f"{user.first_name.capitalize()} "
                 f"{user.last_name.capitalize()}")

    if not is_user_active(user):
        return f"User {full_name} is not active."

    dob = user.date_of_birth
    birth_day, birth_month, birth_year = map(int, dob.split("."))
    age = calculate_age(birth_day, birth_month, birth_year)

    return str({
        "name": full_name,
        "age": age,
        "status": "Active"
    })
```

Exercise [5 min]

Refactor this function:
`demo/functions/01_process_user_data.py`.

Function arguments

→ The ideal number of function arguments is zero.

→ One or two arguments is acceptable.

Function arguments - Bad example

```
def import_project(
    project,
    temp_dir,
    image_dir,
    image_meta_dir,
    project_image_dir,
    project_annotation_dir,
    segment_dir,
    models_dir,
    checkpoints_dir,
    project_model_dir,
    project_checkpoint_dir,
    annotation_dir,
    annotation_classes_dir,
    task,
):
    ...
```

Don't use flag arguments - Bad example

```
def calculate_salary(employee, is_comissioned: bool) → Salary:
    if is_comissioned:
        salary = calculate_comissioned_salary(employee)
    else:
        salary = calculate_full_time_salary(employee)
    return salary
```

Don't use flag arguments - Refactored

```
def calculate_salary(employee: Employee) → Salary:
    return employee.calculate_salary()
```

No side effects

→ Side effects are lies.

→ If your function has no return, there are good chances that it has a side effect.

No side effects - Bad example

```
import pandas as pd
from datetime import date

# Global dataframe
users = pd.DataFrame({
    "name": ["John", "Adam"],
    "surname": ["Doe", "Smith"],
    "birthday": [date(2000, 3, 1), date(1998, 5, 20)]
})
```

No side effects - Bad example

```
import pandas as pd
from datetime import date


# Global dataframe
users = pd.DataFrame({
    "name": ["John", "Adam"],
    "surname": ["Doe", "Smith"],
    "birthday": [date(2000, 3, 1), date(1998, 5, 20)]
})
```

No side effects - Bad example

```
def calculate_user_age(users: pd.DataFrame):
    today = date.today()
    users["birthday"] = users["birthday"].apply(
        lambda bday: today.year
        - bday.year
        - ((today.month, today.day) < (bday.month, bday.day))
    )
```

No side effects - Refactored

```
import pandas as pd
from datetime import date


def add_current_age_column(users: pd.DataFrame) → pd.DataFrame:
    today = date.today()
    users_copy = users.copy()
    users_copy["age"] = users_copy["birthday"].apply(
        lambda birthday: calculate_age_from_birthday(birthday, today)
    )
    return users_copy
```

No side effects - Refactored

```python
import pandas as pd
from datetime import date


def add_current_age_column(users: pd.DataFrame) -> pd.DataFrame:
    today = date.today()
    users_copy = users.copy()
    users_copy["age"] = users_copy["birthday"].apply(
        lambda birthday: calculate_age_from_birthday(birthday, today)
    )
    return users_copy
```

No side effects - Refactored

```
def calculate_age_from_birthday(birthday: date, today: date):
    has_had_birthday_this_year = (today.month, today.day) < (
        birthday.month,
        birthday.day,
    )
    years_difference = today.year - birthday.year
    age = years_difference - has_had_birthday_this_year
    return age
```

No side effects? An example from sklearn

```
from sklearn.preprocessing import StandardScaler



# Initialize the StandardScaler
standard_scaler = StandardScaler()

print(standard_scaler.__dict__)
```

No side effects? An example from sklearn

```
>>> {'with_mean': True,
     'with_std': True,
     'copy': True}
```

No side effects? An example from sklearn

```
# Sample data
feature_samples = [[-1, 2], [-0.5, 6], [0, 10], [1, 18]]

# Fit the scaler to the samples
standard_scaler.fit(feature_samples)

print(standard_scaler.__dict__)
```

No side effects? An example from sklearn

```
>>> {'with_mean': True, 'with_std': True, 'copy': True,
    'n_features_in_': 2, 'n_samples_seen_': np.int64(4),
    'mean_': array([-0.125, 9.]), 'var_': array([0.546875, 35.]),
    'scale_': array([0.73950997, 5.91607978])}
```

Data classes

Data classes in Python

→ Create a class that inherits from `typing.NamedTuple`

→ Decorate your class with `@dataclasses.dataclass`

Data class with `typing.NamedTuple`

```
from typing import NamedTuple


class Coordinates(NamedTuple):
    latitude: float
    longitude: float
```

Data class with `@dataclasses.dataclass`

```python
from dataclasses import dataclass


@dataclass
class User:
    first_name: str
    last_name: str
    is_active: bool
```

Feature	typing.NamedTuple	@dataclasses.dataclass
Introduced in	Python 3.5	Python 3.7
Mutability	Immutable	Mutable by default (immutable with `frozen=True`)
Type Checking	Yes (using annotations)	Yes (using annotations)
Field Defaults	No	Yes (can have default values or factory functions)
Inheritance	Must inherit from `NamedTuple`	Inheritance possible but not necessary
Custom Methods	Can define methods but need to use `_fields` to access fields within methods	Can define methods and directly access fields
Optional Parameters (init=False, repr=False, etc.)	Limited (Cannot easily exclude fields from the constructor or representation)	Flexible (e.g., `init=False`, `repr=False`, `compare=False`)
Equality and Comparison	Automatically supports equality comparison	Supports equality and comparison operators (customizable with `order=True`)
Automatic Representation	Provides a readable `__repr__` by default	Provides a customizable `__repr__` by default
Field Metadata	No	Yes (using `field(metadata={ ... })`)
Performance	Slightly more efficient due to immutability and simpler structure	Less efficient due to additional features like mutability, default values, etc.
Integration with Other Libraries	Limited (less extensible due to immutability)	Highly extensible, compatible with many third-party libraries like Marshmallow, Pydantic, etc.
Usage Recommendation	Best for simple, fixed collections of data with named fields	Best for more complex data structures where mutability, defaults, or additional functionality are needed

Data class as code smell

Data classes are often a sign of behavior in the wrong place.
— Martin Fowler and Kent Beck[3]

[3] *Refactoring: Improving the Design of Existing Code, 2nd edition*

Interfaces

What are interfaces?

→ In Go: An interface **type** is defined as a set of **method signatures.**

→ In Java: An abstract **type** that is used to declare a **behavior** that classes **must implement.**

Interface example in Go

```
import (
    "fmt"
    "math"
)

type geometry interface {
    area() float64
    perim() float64
}

type circle struct {
    radius float64
}

func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}

func (c circle) perim() float64 {
    return 2 * math.Pi * c.radius
}
```

Interface example in Go

```
import (
    "fmt"
    "math"
)

type geometry interface {
    area() float64
    perim() float64
}

type circle struct {
    radius float64
}

func (c circle) area() float64 {
    return math.Pi * c.radius * c.radius
}

func (c circle) perim() float64 {
    return 2 * math.Pi * c.radius
}
```

Duck typing in Python

Types are defined by their supported operations.
— Luciano Ramalho[4]

[4] *Fluent Python, 2nd edition*

Duck typing in Python – an example

```
class Artist:
    def draw(self):
        print("Draw a painting.")

class Lottery:
    def draw(self):
        print("Draw lottery numbers.")
```

Duck typing in Python – an example

The type of `painer` is irrelevant, as long as it has a method `draw`:

```
def draw_a_painting(painter):
    painter.draw()
```

Duck typing in Python – an example

This is valid:

```
artist = Artist()
draw_a_painting(artist)
```

```
>>> Draw a painting.
```

Duck typing in Python – an example

And this is also valid:

```
lottery = Lottery()
draw_a_painting(lottery)
```

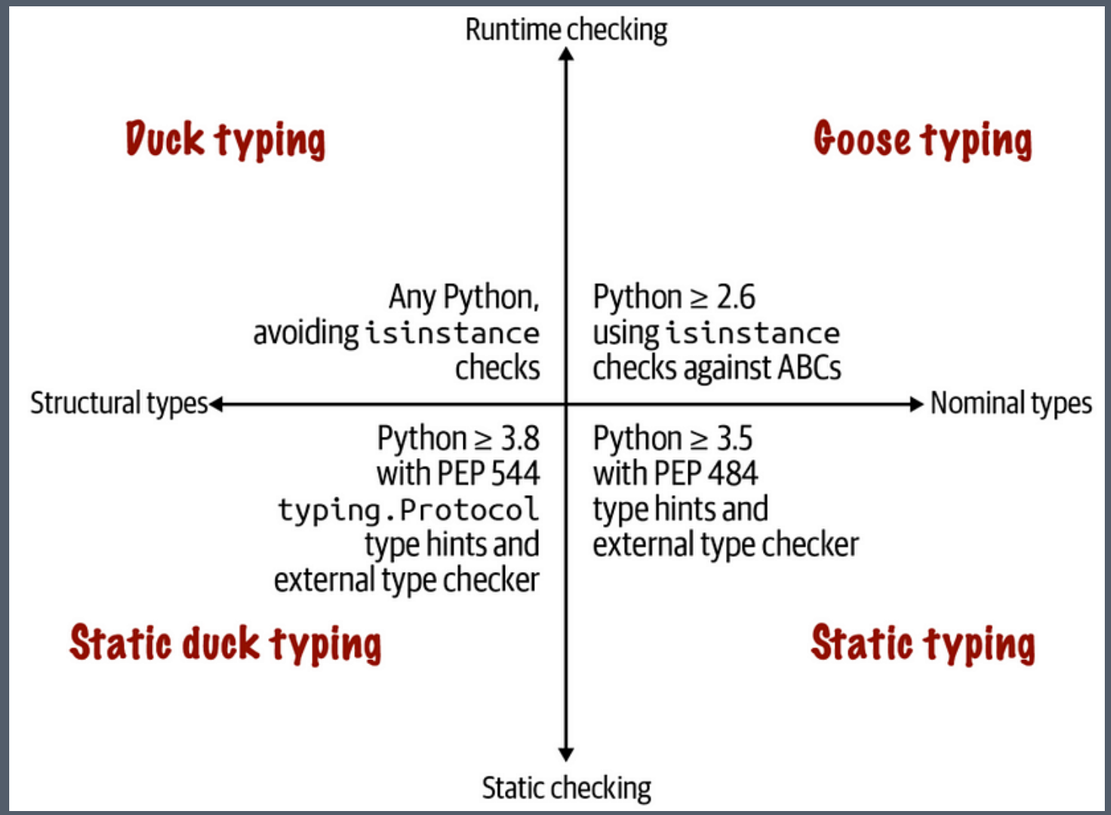
```
>>> Draw lottery numbers.
```

Duck typing in Python – an example

```
isinstance(artist, Lottery)
isinstance(lottery, Artist)
```

```
>>> False
>>> False
```

Interfaces in Python – 4 ways[5]

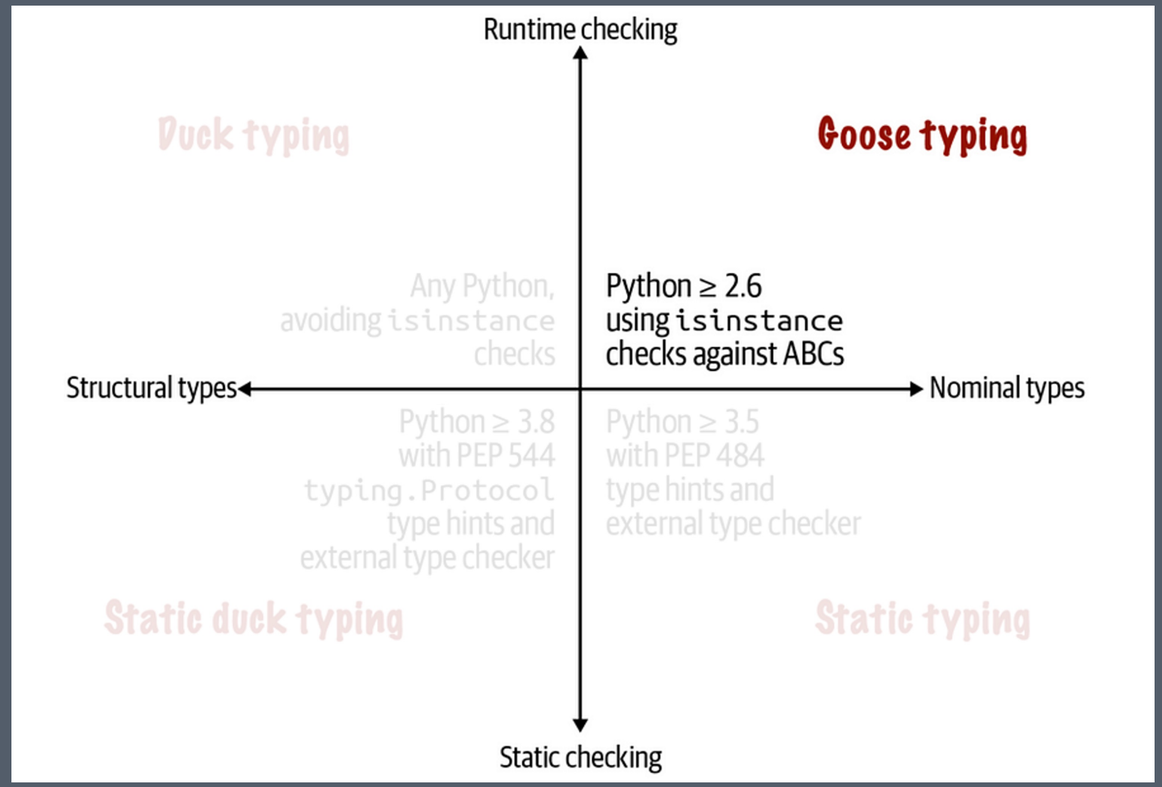


[5] Advice by Alex Martelli, *Fluent Python, 2nd edition*

Interfaces with goose typing

→ Subclass from an abstract base class (ABC) or

→ register your class as a virtual subclass of the ABC.



Interfaces with goose typing

Works even without explicit subclassing:*

```
from collections import abc

class SizedLikeClass:
    def __len__(self):
        ...

isinstance(SizedLikeClass(), abc.Sized)
```

```
>>> True
```

* But don't do it.

Goose typing - subclassing an ABC

→ Subclass from an existing ABC before inventing your own[5].

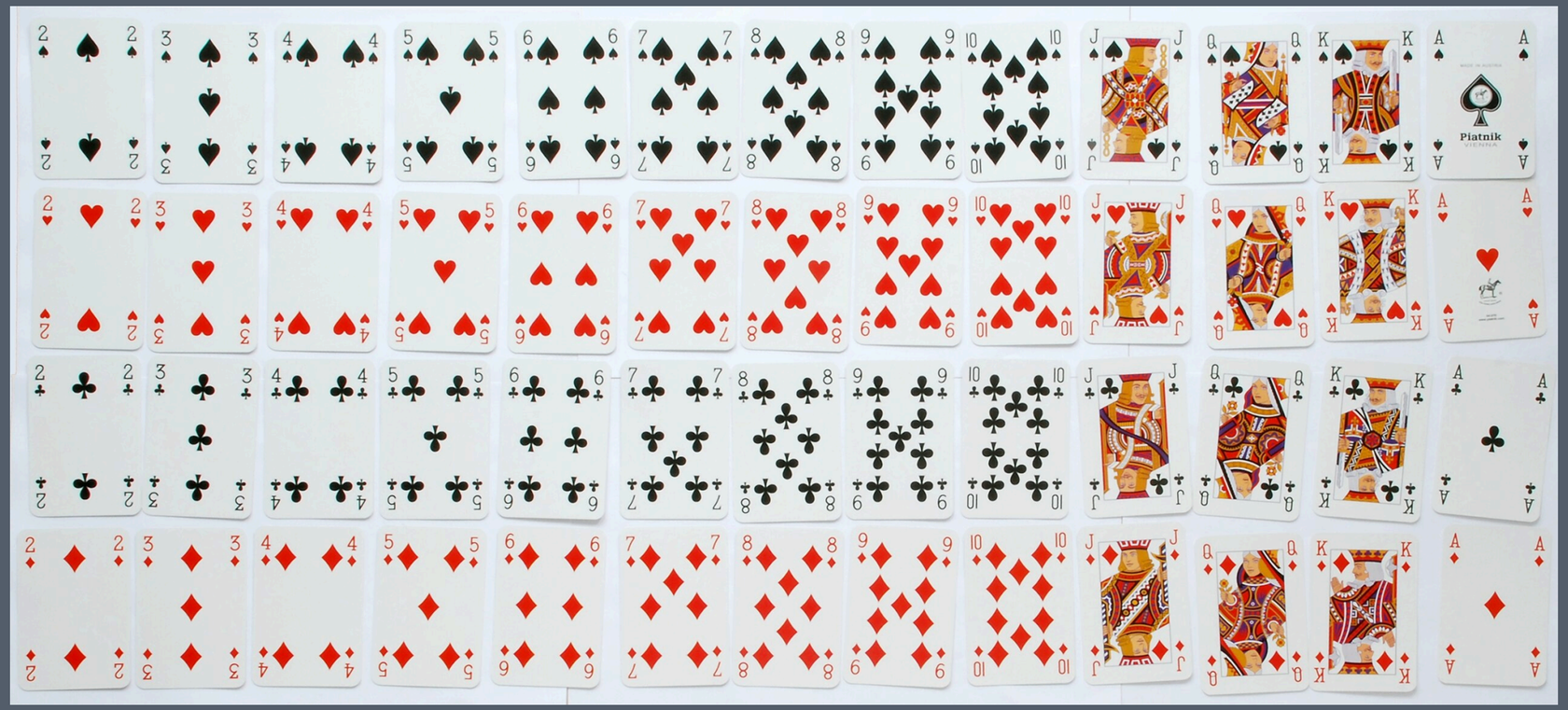
→ Use an existing ABC from `collections.abc`.

→ If you need to create a custom ABC, use the `abc` module.

[5] Advice by Alex Martelli, *Fluent Python, 2nd edition*

Goose typing - subclassing an ABC

Imagine that you want to create a class that holds the French card deck.



Goose typing - subclassing an ABC

```
from collections import abc


print(abc.MutableSequence.__abstractmethods__)
```

```
>>> frozenset({'__len__', '__getitem__',
    '__setitem__', 'insert', '__delitem__'})
```

Goose typing - subclassing an ABC

```
from collections import abc
from typing import NamedTuple



class Card(NamedTuple):
    rank: str
    suit: str
```

Goose typing - subclassing an ABC

```
class CardDeck(abc.MutableSequence):
    ranks = [str(rank) for rank in range(2, 11)] + list("JQKA")
    suits = ["clubs", "spades", "diamonds", "hearts"]

    def __init__(self):
        self._cards = [Card(rank, suit) for rank in self.ranks
                       for suit in self.suits]

    def __getitem__(self, position):
        return self._cards[position]
```

Goose typing - subclassing an ABC

```
class CardDeck(abc.MutableSequence):
    ranks = [str(rank) for rank in range(2, 11)] + list("JQKA")
    suits = ["clubs", "spades", "diamonds", "hearts"]

    def __init__(self):
        self._cards = [Card(rank, suit) for rank in self.ranks
                       for suit in self.suits]

    def __getitem__(self, position):
        return self._cards[position]
```

Goose typing - subclassing an ABC

```
    def __setitem__(self, position, value):
        self._cards[position] = value

    def __len__(self):
        return len(self._cards)

    def __delitem__(self, position):
        del self._cards[position]

    def insert(self, position, value):
        self._cards.insert(position, value)
```

Goose typing - subclassing an ABC

```
    def __setitem__(self, position, value):
        self._cards[position] = value

    def __len__(self):
        return len(self._cards)

    def __delitem__(self, position):
        del self._cards[position]

    def insert(self, position, value):
        self._cards.insert(position, value)
```

Goose typing - register as virtual subclass of an ABC

- → `register` your class as a *virtual subclass* of an ABC.
- → No inheritance from the ABC.
- → Static type checkers can't handle virtual subclassing (yet).

Goose typing - register as virtual subclass of an ABC

```
from collections.abc import MutableSequence

@MutableSequence.register
class CardDeck:
    ranks = [str(rank) for rank in range(2, 11)] + list("JQKA")
    suits = ["clubs", "spades", "diamonds", "hearts"]
    # Implement __len__, __getitem__, __setitem__, insert, __delitem__ analogously
    ...

# Equivalent to the decorator:
# MutableSequence.register(CardDeck)
```

Goose typing - register as virtual subclass of an ABC

`CardDeck` passes the `isinstance` and `issubclass` checks:

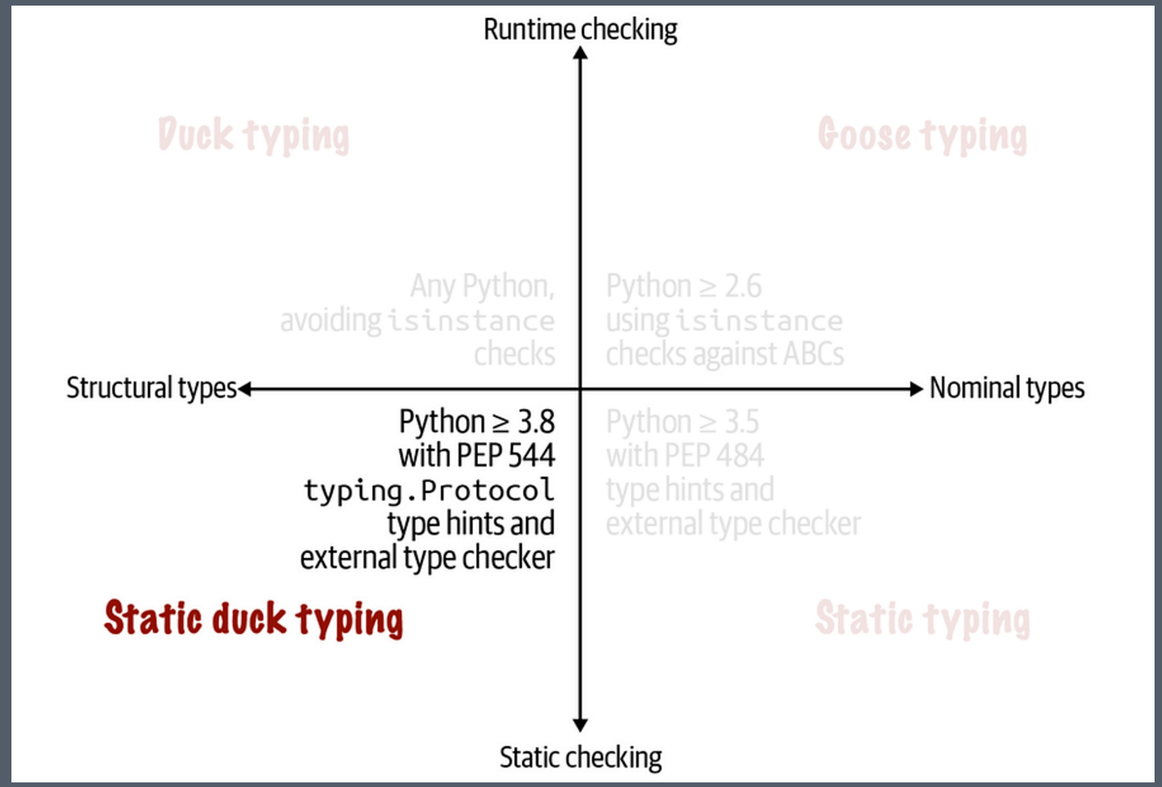
```
card_deck = CardDeck()
isinstance(card_deck, abc.MutableSequence)
issubclass(CardDeck, abc.MutableSequence)
```

```
>>> True
>>> True
```

Static duck typing

→ Static checking approach.

→ Define interfaces with static protocols in `typing.Protocol`.



Static duck typing - create a static protocol

```
from typing import Any, Protocol, runtime_checkable


@runtime_checkable
class RandomPicker(Protocol):
    def pick(self) → Any:
        pass
```

Static duck typing - create a static protocol

```
import random
from typing import Iterable

class SimplePicker:
    def __init__(self, items: Iterable) -> None:
        self._items = list(items)
        random.shuffle(self._items)

    def pick(self) -> Iterable:
        return self._items.pop()
```

Static duck typing - create a static protocol

```
simple_picker = SimplePicker(items=[1, 5, 7, 20])

isinstance(simple_picker, RandomPicker)
issubclass(SimplePicker, RandomPicker)
```

```
>>> True
>>> True
```

Classes

How big should classes be?

→ Classes should be small,

→ and then they should be smaller than that.

Classes should be small

The SOLID design principles

- **S** Single Responsibility Principle
- **O** Open Closed Principle
- **L** Liskov Substitution Principle
- **I** Interface Segregation Principle
- **D** Dependency Inversion Principle

Single Responsibility Principle[7]

A class or module should have one and only one reason to change - one responsibility.

[7] Some of these fun SOLID memes come from here.

Single Responsibility Principle - Bad example

```python
from sklearn.linear_model import LinearRegression


class Model:

    def __init__(self, data: pd.DataFrame):
        self.data = data
        self.model = None

    def train(self) -> None:
        self.data = (self.data - self.data.mean()) / self.data.std()
        features = self.data.columns.difference(["target"])
        self.model = LinearRegression().fit(self.data[features], self.data["target"])

    def evaluate(self) -> float:
        features = self.data.columns.difference(["target"])
        predictions = self.model.predict(self.data[features])
        mean_squared_error = ((predictions - self.data["target"]) ** 2).mean()
        return mean_squared_error
```

Exercise [5 min]

Refactor this class:
`demo/classes/01_linear_regression.py`.

Cohesion

→ Classes should have high cohesion.

→ Classes should have a small number of instance variables.

A bad example of low cohesion

```
class MultimediaManager:
    def __init__(self, video_files, audio_files):
        self.video_files = video_files
        self.audio_files = audio_files

    def play_video(self):
        for video in self.video_files:
            print(f"Playing video: {video}")

    def play_audio(self):
        for audio in self.audio_files:
            print(f"Playing audio: {audio}")
```

A bad example of low cohesion - Refactored

```
class VideoPlayer:
    def __init__(self, video_files):
        self.video_files = video_files

    def play_video(self):
        for video in self.video_files:
            print(f"Playing video: {video}")
```

A bad example of low cohesion - Refactored

```
class AudioPlayer:
    def __init__(self, audio_files):
        self.audio_files = audio_files

    def play_audio(self):
        for audio in self.audio_files:
            print(f"Playing audio: {audio}")
```

Open Closed Principle

A class or module should be open for extension but closed for modification.

Open Closed Principle - Bad example

```
from typing import Any


class SqlQuery:
    """Generates SQL query string."""

    def __init__(self, table_name: str, columns: list[Column]):
        self.table = table_name
        self.columns = columns
```

Open Closed Principle - Bad example

```
    def create(self):
        """Generates an SQL create string."""
        ...

    def insert(self, records: list[Any]):
        """Generates an SQL insert data string."""
        ...

    def select(self, column: Column, pattern: str):
        """Generates an SQL select all records string."""
        ...

    def _select_with_criteria(self, criteria: str):
        """Generates an SQL select by criteria string."""
        ...
```

Open Closed Principle - Refactored

```python
from abc import ABC, abstractmethod


class Sql(ABC):
    """
    Abstract base class for SQL query string representations.
    This class contains no implementation and is not intended to be used on its own.
    """
    def __init__(self, table_name: str, columns: list[Column]):
        self.table = table_name
        self.columns = columns

    @abstractmethod
    def generate(self) -> str:
        pass
```

Open Closed Principle - Refactored

```
class CreateSql(Sql):

    def generate(self):
        ...


class InsertSql(Sql):

    def __init__(self, table_name: str, columns: list[Column], records: list[Any]):
        super().__init__(table_name, columns)
        self.records = records

    def generate(self):
        ...
```

Open Closed Principle - Refactored

```
class SelectSql(Sql):

    def generate(self):
        ...


class SelectWithCriteriaSql(Sql):

    def __init__(self, table_name: str, columns: list[Column], criteria: Criteria):
        super().__init__(table_name, columns)
        self.criteria = criteria

    def generate(self):
        ...
```

Open Closed Principle and Polymorphism

Abstraction is the key to the Open Closed Principle.

Open Closed Principle and Polymorphism - Bad example

```
from enum import Enum
from sklearn.metrics import accuracy_score, r2_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LinearRegression


class SupportedModels(Enum):
    classifier = DecisionTreeClassifier
    regressor = LinearRegression
```

Open Closed Principle and Polymorphism - Bad example

```
class Model:

    def __init__(self, model):
        self.model = model

    def calculate_metric(self, y_true, y_pred):
        """Calculate metrics for a given model."""
        if isinstance(self.model, SupportedModels.classifier.value):
            metric = accuracy_score(y_true, y_pred)
        elif isinstance(self.model, SupportedModels.regressor.value):
            metric = r2_score(y_true, y_pred)
        else:
            raise NotImplementedError(f"Metrics for {self.model.__class__.__name__} "
                                      f"are not implemented.")
        return metric
```

Open Closed Principle and Polymorphism - Bad example

```
class Model:

    def __init__(self, model):
        self.model = model

    def calculate_metric(self, y_true, y_pred):
        """Calculate metrics for a given model."""
        if isinstance(self.model, SupportedModels.classifier.value):
            metric = accuracy_score(y_true, y_pred)
        elif isinstance(self.model, SupportedModels.regressor.value):
            metric = r2_score(y_true, y_pred)
        else:
            raise NotImplementedError(f"Metrics for {self.model.__class__.__name__} "
                                      f"are not implemented.")
        return metric
```

Open Closed Principle and Polymorphism - Bad example

```
class Model:

    def __init__(self, model):
        self.model = model

    def calculate_metric(self, y_true, y_pred):
        """Calculate metrics for a given model."""
        if isinstance(self.model, SupportedModels.classifier.value):
            metric = accuracy_score(y_true, y_pred)
        elif isinstance(self.model, SupportedModels.regressor.value):
            metric = r2_score(y_true, y_pred)
        else:
            raise NotImplementedError(f"Metrics for {self.model.__class__.__name__} "
                                      f"are not implemented.")
        return metric
```

Exercise [5 min]

Refactor this class:
`demo/classes/02_calculate_metric.py`.

A bad example from the Requests library

```
# A Session class code snippet from the requests library

class Session(SessionRedirectMixin):
    """A Requests session."""

    def __init__(self):
        self.headers = default_headers()
        self.auth = None
        self.proxies = {}
        self.hooks = default_hooks()
        self.params = {}
        self.stream = False
        self.verify = True
        self.cert = None
        self.max_redirects = DEFAULT_REDIRECT_LIMIT
        self.trust_env = True
        self.cookies = cookiejar_from_dict({})
        self.adapters = OrderedDict()
        self.mount("https://", HTTPAdapter())
        self.mount("http://", HTTPAdapter())
```

```
def prepare_request(self, request):
    ...

def request(
        self,
        method,
        url,
        params=None,
        data=None,
        headers=None,
        cookies=None,
        files=None,
        auth=None,
        timeout=None,
        allow_redirects=True,
        proxies=None,
        hooks=None,
        stream=None,
        verify=None,
        cert=None,
        json=None,
):
    """Constructs a :class:`Request <Request>`, prepares it and sends it."""
    req = Request(
        method=method.upper(),
        url=url,
        headers=headers,
        files=files,
        data=data or {},
        json=json,
        params=params or {},
        auth=auth,
        cookies=cookies,
        hooks=hooks,
    )
    ...
```

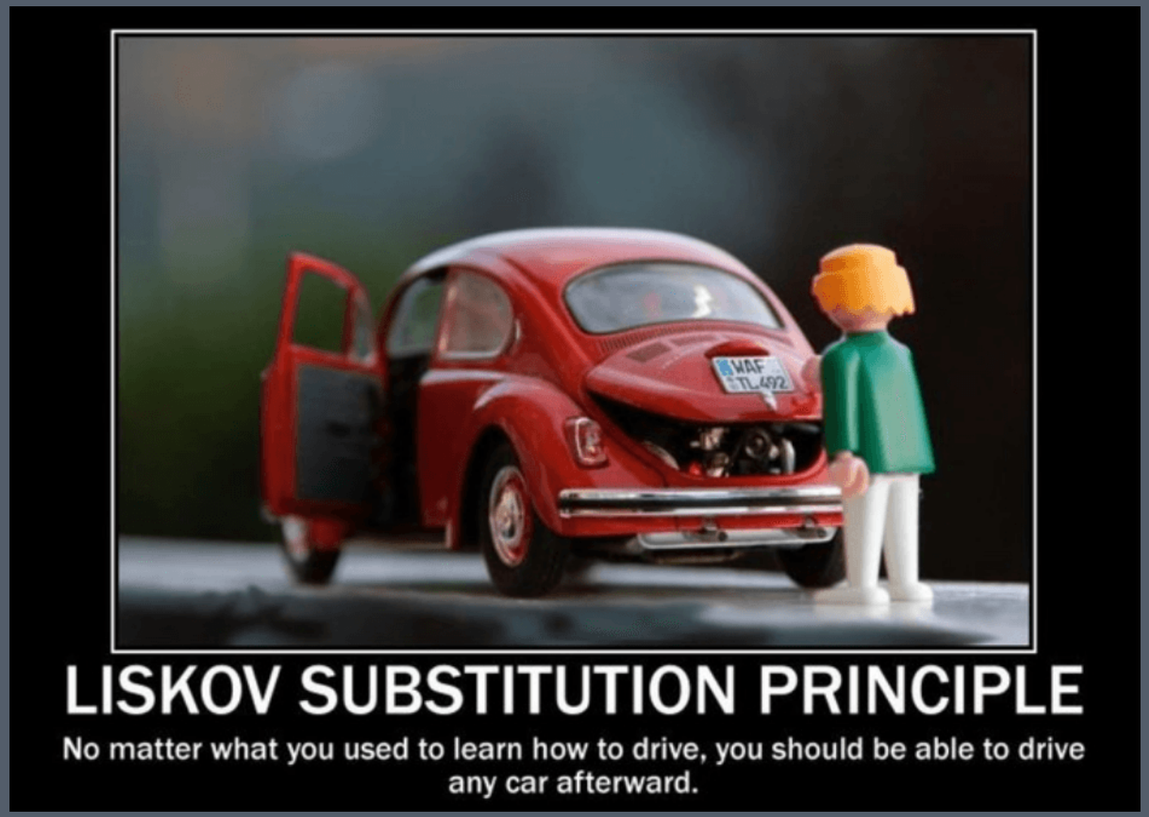
```
def post(self, url, data=None, json=None, **kwargs):
    """Sends a POST request."""
    ...

def get(self, url, **kwargs):
    """Sends a GET request."""
    ...

def send(self, request, **kwargs):
    """Send a given PreparedRequest."""
    ...
```

Liskov Substitution Principle

Subclasses should be substitutable for their base classes.



Liskov Substitution Principle - Good example

```python
from dataclasses import dataclass


@dataclass
class Employee:
    name: str
    salary: float


@dataclass
class SoftwareDeveloper(Employee):
    years_of_experience: int
```

Liskov Substitution Principle - Good example

```
def get_employee_salary(employee: Employee):
    print(f"Employee {employee.name} earns "
          f"{str(employee.salary)}.")
    return employee.salary
```

Liskov Substitution Principle - Good example

```
employee_john = Employee("John", 120000)

get_employee_salary(employee_john)
```

```
>>> Employee John earns 120000.
```

Liskov Substitution Principle - Good example

```
software_developer_ivy = SoftwareDeveloper("Ivy", 150000, 3)

get_employee_salary(software_developer_ivy)
```

```
>>> Employee Ivy earns 150000.
```

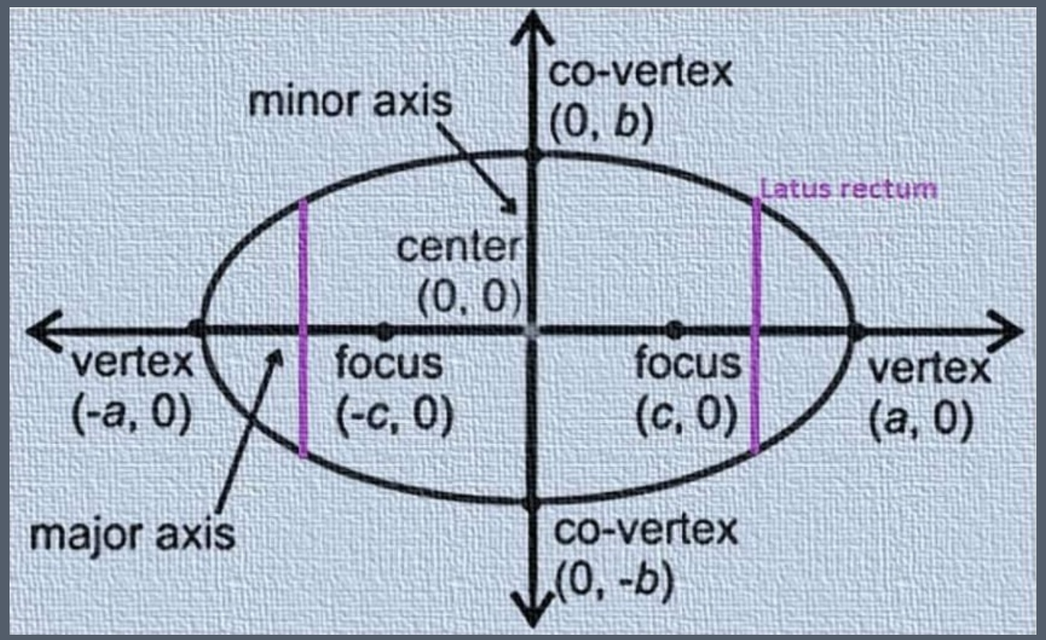
Dependency Inversion Principle

Depend upon abstractions. Do not depend upon concretions.

The Ellipse / Circle dilemma

An ellipse in Carthesian coordinates with a center $(0, 0)$ is defined by:

- a first focus point $(0, -c)$, where $c \in \mathbb{R}^+$,
- a second focus point $(0, c)$, where $c \in \mathbb{R}^+$,
- a semi-major axis a, where $a \in \mathbb{R}^+$,
- a semi-minor axis b, where $b \in \mathbb{R}^+$.



The Ellipse / Circle dilemma

The following holds for the foci and the semi-major and semi-minor axes: $c^2 = a^2 - b^2$.

The area of the ellipse: πab

The perimeter of the ellipse: $2\pi\sqrt{\frac{a^2 + b^2}{2}}$

The Ellipse / Circle dilemma - `class Ellipse`

```
class Ellipse:

    def __init__(
            self,
            semi_major_axis: float,
            semi_minor_axis: float,
    ):
        self._semi_major_axis = semi_major_axis
        self._semi_minor_axis = semi_minor_axis
```

The Ellipse / Circle dilemma - `class Ellipse`

```python
    @property
    def semi_major_axis(self):
        return self._semi_major_axis

    @property
    def semi_minor_axis(self):
        return self._semi_minor_axis

    @property
    def area(self) -> float:
        area = math.pi * self._semi_major_axis * self._semi_minor_axis
        return area
```

The Ellipse / Circle dilemma - `class Ellipse`

```
    def set_semi_major_axis(self, semi_major_axis: float):
        self._semi_major_axis = semi_major_axis

    def set_semi_minor_axis(self, semi_minor_axis: float):
        self._semi_minor_axis = semi_minor_axis
```

The Ellipse / Circle dilemma

→ The circle is a special case of an ellipse with radius $r := a = b$.

→ This tempts us to model the classes inheritance like this:

The Ellipse / Circle dilemma - `class Circle`

```
class Circle(Ellipse):

    def __init__(self, radius: float):
        super().__init__(radius, radius)
```

The Ellipse / Circle dilemma - Users ruin everything

A user of `Ellipse` implements this function:

```
def double_ellipse_semi_major_axis(ellipse: Ellipse):
    ellipse.set_semi_major_axis(ellipse.semi_major_axis * 2)
```

The Ellipse / Circle dilemma - Users ruin everything

```
e = Ellipse(3, 2)
print(e)
double_ellipse_semi_major_axis(e)
print(e)
print(f"Is ellipse? {isinstance(e, Ellipse)}")
```

```
>>> Ellipse(semi-major-axis: 3, semi-minor-axis: 2)
>>> Ellipse(semi-major-axis: 6, semi-minor-axis: 2)
>>> Is ellipse? True
```

The Ellipse / Circle dilemma - Users ruin everything

```
c = Circle(3)
print(c)
double_ellipse_semi_major_axis(c)
print(c)
print(f"Is circle? {isinstance(c, Circle)}")
```

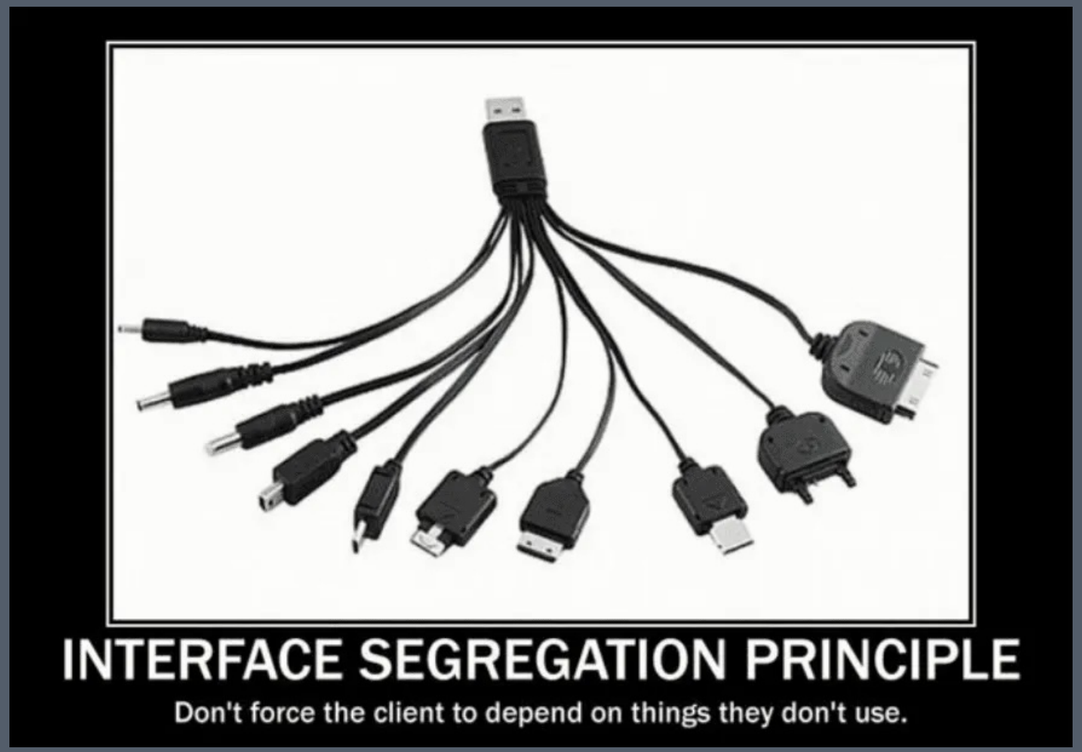
```
>>> Circle(semi-major-axis: 3, semi-minor-axis: 3)
>>> Circle(semi-major-axis: 6, semi-minor-axis: 3)
>>> Is circle? True
```

Exercise [10 min]

Refactor this implementation:
`demo/classes/03_circle_ellipse.py`.

Interface Segregation Principle

Many specific interfaces are better than one general purpose interface.



Interface Segregation Principle - A very general interface (bad)

```
from abc import ABC, abstractmethod


class UserActions(ABC):
    @abstractmethod
    def create_post(self):
        pass

    @abstractmethod
    def edit_post(self):
        pass
```

Interface Segregation Principle - A very general interface (bad)

```
    @abstractmethod
    def delete_post(self):
        pass

    @abstractmethod
    def view_post(self):
        pass

    @abstractmethod
    def manage_users(self):
        pass
```

Interface Segregation Principle - A user

```
class Viewer(UserActions):
    def create_post(self):
        raise NotImplementedError("Viewer cannot create posts")

    def edit_post(self):
        raise NotImplementedError("Viewer cannot edit posts")

    def delete_post(self):
        raise NotImplementedError("Viewer cannot delete posts")

    def view_post(self):
        print("Viewer views a post")

    def manage_users(self):
        raise NotImplementedError("Viewer cannot manage users")
```

Interface Segregation Principle - A user

```
class Viewer(UserActions):
    def create_post(self):
        raise NotImplementedError("Viewer cannot create posts")

    def edit_post(self):
        raise NotImplementedError("Viewer cannot edit posts")

    def delete_post(self):
        raise NotImplementedError("Viewer cannot delete posts")

    def view_post(self):
        print("Viewer views a post")

    def manage_users(self):
        raise NotImplementedError("Viewer cannot manage users")
```

Interface Segregation Principle - Many small interfaces (better)

```
from abc import ABC, abstractmethod


class PostCreator(ABC):
    @abstractmethod
    def create_post(self):
        pass

class PostEditor(ABC):
    @abstractmethod
    def edit_post(self):
        pass
```

Interface Segregation Principle - Many small interfaces (better)

```
class PostDeletor(ABC):
    @abstractmethod
    def delete_post(self):
        pass

class PostViewer(ABC):
    @abstractmethod
    def view_post(self):
        pass

class UsersManager(ABC):
    @abstractmethod
    def manage_users(self):
        pass
```

Interface Segregation Principle - A happier user

```
class Viewer(PostViewer):
    def view_post(self):
        print("Viewer views a post")
```

Most important concepts

- → Pick intention-revealing names.
- → Functions should be small and should do only one thing.
- → Try not to invent your own abstract base classes.
- → When designing classes, remember SOLID.

Q & A