Coding to Interfaces: structural subtyping in Python

Carlo Bertini & Francesco Panico

Fiscozen

Chi siamo

Carlo Bertini

Senior software engineer @Fiscozen

https://github.com/WaYdotNET

Francesco Panico

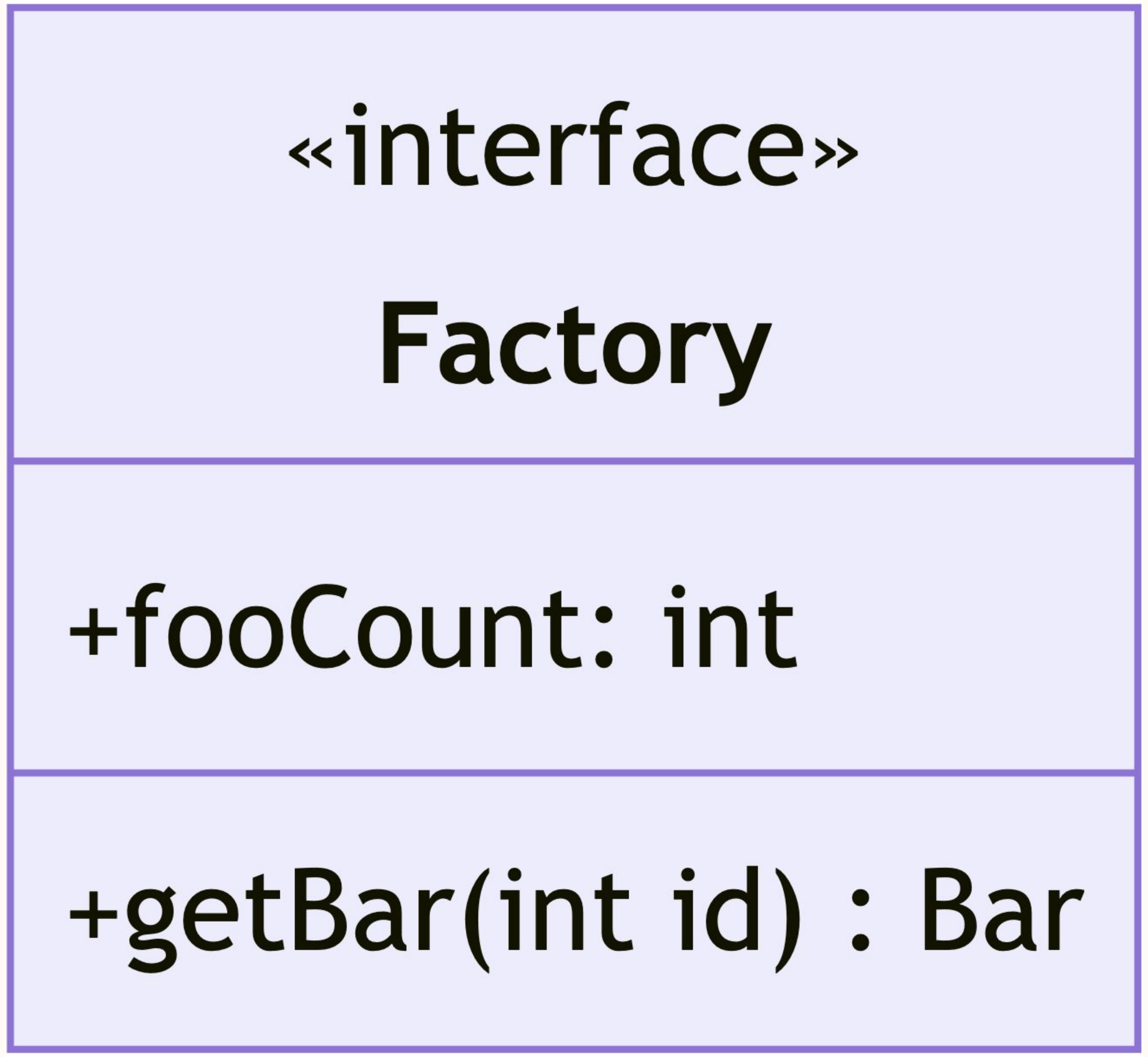
Tech team leader @Fiscozen

https://github.com/panicofr

Sommario

- Cos’è un’interfaccia e come usarla per migliorare il design del codice
- Type Hints in Python
- L’analisi statica del codice
- Nominal e Structural subtyping
- Abstract Base Classes
- Protocols
- Domande

Che cosa è un’interfaccia ?



Cos’è il “Coding to interfaces”

Codice utilizzato come esempio

```python
def print_name(user):
    print(user.full_name(), user.tags)
```

Esempio 1

```python
class BusinessUser:
    first_name = None
    last_name = None
    tags = None

    def __init__(self, first_name, last_name, tags):
        self.first_name = first_name
        self.last_name = last_name
        self.tags = tags

    def full_name(self):
        return f"{self.first_name} - {self.last_name}"
```

Esempio 1

```python
business_user = BusinessUser(first_name="BU", last_name="Bar", tags=["red", "white"])


def print_name(obj):
   print(obj.full_name(), obj.tags)


print_name(business_user)
```

Esempio 1

```python
business_user = BusinessUser(first_name="BU", last_name="Bar", tags=["red", "white"])

def print_name(obj):
   print(obj.full_name(), obj.tags)

print_name(business_user)
# -> BU - Bar ['red', 'white']
```

Esempio 1

```python
def print_name(obj):
    print(obj.full_name(), obj.tags)


print_name(None)
```

Esempio 1

```python
def print_name(obj):
    print(obj.full_name(), obj.tags)


print_name(None)
# -> AttributeError: 'NoneType' object has no attribute 'full_name'
```

L’analisi statica

Tool di analisi statica

- mypy -> https://mypy-lang.org/
- pytype -> https://google.github.io/pytype/
- pyright -> https://github.com/Microsoft/pyright
- pyre -> https://pyre-check.org/

Analisi statica e runtime

```python
class BusinessUser:
    first_name = None
    last_name = None
    tags

    def __init__(self, first_name, last_name, tags):
        self.first_name = first_name
        self.last_name = last_name
        self.tags = tags

    def full_name(self):
        return f"{self.first_name} - {self.last_name}"


business_user = BusinessUser(first_name="BU", last_name="Bar", tags=["red", "white"])

def print_name(obj):
    print(obj.full_name(), obj.tags)

print_name(business_user)
print_name(None)
```

Analisi statica e runtime

```python
print_name(business_user)
# -> BU - Bar ['red', 'white']

print_name(None)
# -> AttributeError: 'NoneType' object has no attribute 'full_name'
```

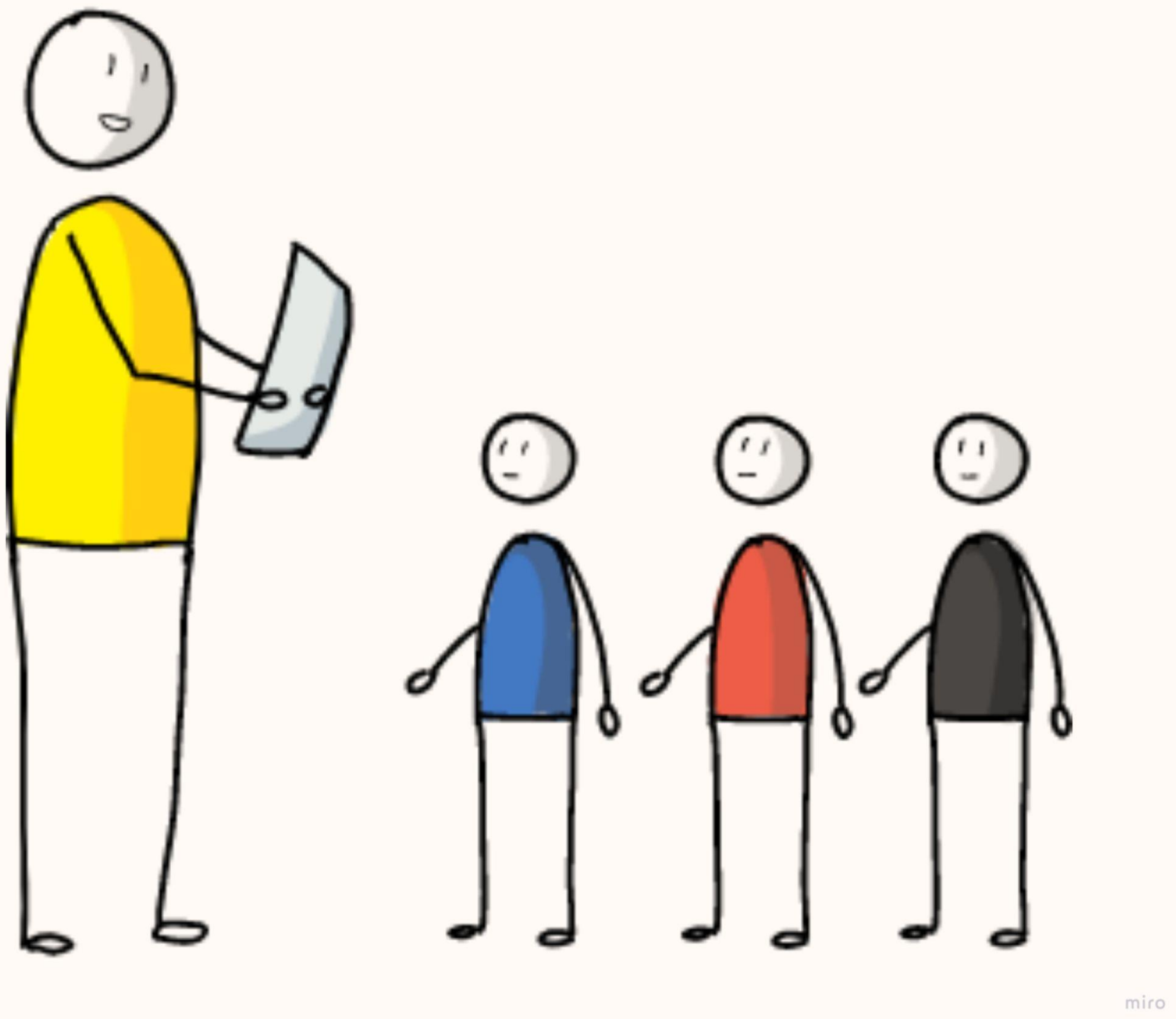
```
❯ mypy 02.py
```

Analisi statica e runtime

```python
print_name(business_user)
# -> BU - Bar ['red', 'white']

print_name(None)
# -> AttributeError: 'NoneType' object has no attribute 'full_name'
```

```
❯ mypy 02.py
Success: no issues found in 1 source file
```

Introduzione ai type hints (PEP 484)



Type hints: vantaggi e svantaggi

Vantaggi:

- Type checking statico: consente di prevenire alcuni bug (grazie a tools di analisi statica)
- Maggiore leggibilità del codice e documentazione “implicita”
- Miglioramento dell’autocompletamento degli IDE

Svantaggi

- Sintassi da implementare
- Codice più prolisso
- Difficoltà nella tipizzazione di una codebase esistente
- Utilizzo librerie di terze parti prive di type hints (o stubs)

Analisi statica e runtime - con i type hints

```python
class BusinessUser:
    first_name: str | None = None
    last_name: str | None = None
    tags: list[str]

    def __init__(self, first_name: str, last_name: str, tags: list[str]) -> None:
        self.first_name = first_name
        self.last_name = last_name
        self.tags = tags

    def full_name(self) -> str:
        return f"{self.first_name} - {self.last_name}"


def print_name(obj: BusinessUser) -> None:
    print(obj.full_name(), obj.tags)
```

Analisi statica e runtime - con i type hints

```python
class BusinessUser:
    first_name: str | None = None      # typing.Union[str,None] with python < 3.10
    last_name: str | None = None
    tags: list[str]

    def __init__(self, first_name: str, last_name: str, tags: list[str]) -> None:
        self.first_name = first_name
        self.last_name = last_name
        self.tags = tags

    def full_name(self) -> str:
        return f"{self.first_name} - {self.last_name}"


def print_name(obj: BusinessUser) -> None:
    print(obj.full_name(), obj.tags)
```

Analisi statica e runtime - con i type hints

```python
print_name(business_user)
# -> BU - Bar ['red', 'white']

print_name(None)
# -> AttributeError: 'NoneType' object has no attribute 'full_name'
```

```
❯ mypy 03.py
```

Analisi statica e runtime - con i type hints

```python
print_name(business_user)
# -> BU - Bar ['red', 'white']

print_name(None)
# -> AttributeError: 'NoneType' object has no attribute 'full_name'
```

```
❯ mypy 03.py
03.py:26: error: Argument 1 to "print_name" has incompatible type "None"; expected "BusinessUser"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

Analisi statica e runtime - con i type hints

```python
# python >= 3.9
TagsType = list[str]


class BusinessUser:
    first_name: str | None = None
    last_name: str | None = None
    tags: TagsType

    def __init__(self, first_name: str, last_name: str, tags: TagsType) -> None:
        # cut
```

```python
# python < 3.9
from typing import List

TagsType = List[str]
```

Coding to interfaces

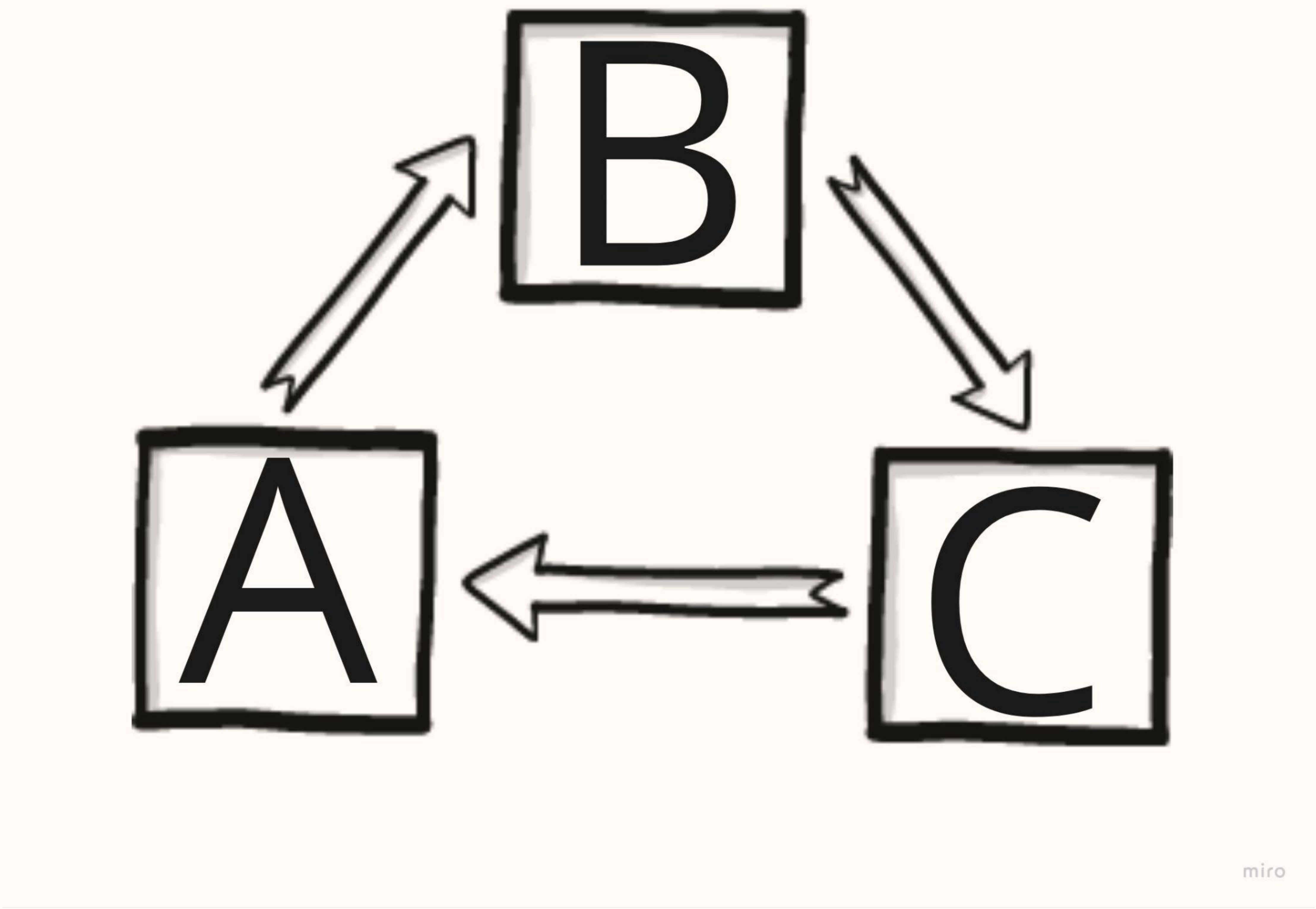
```
interface PrintableObject:
    tags: list[str]
    def full_name(self) -> str: pass
```

Nominal Subtyping

```python
class MyInterface:
    def foo(self) -> str:
        raise NotImplementedError()


class MyClass(MyInterface):
    def foo(self) -> str:
        return "bar"
```

Una classe *B* è sottotipo di una classe *A* se e solo se viene dichiarato esserlo, direttamente o indirettamente

Abstract Base Classes (PEP 3119)



ABC: Nominal Subtyping

```python
import abc


class BaseUser(abc.ABC):
    @property
    @abc.abstractmethod
    def tags(self) -> list[str]: pass

    @abc.abstractmethod
    def full_name(self) -> str: pass
```

```python
class OperatorUser(BaseUser):
    name: str | None = None
    tags: list[str] = ["ops"]

    def __init__(self, name: str) -> None:
        self.name = name

    def full_name(self) -> str:
        return f"{self.name}"
```

ABC: Nominal Subtyping

```python
class BaseUser(abc.ABC):
    # cut

def print_name(obj: BaseUser) -> None:
    print(obj.full_name(), obj.tags)

print_name(business_user)
```

```
❯ mypy 07a.py
```

ABC: Nominal Subtyping

```python
class BaseUser(abc.ABC):
    # cut

def print_name(obj: BaseUser) -> None:
    print(obj.full_name(), obj.tags)

print_name(business_user)
# -> BU - Bar ['red', 'white']
```

```
❯ mypy 07a.py
Success: no issues found in 1 source file
```

E se...

```python
class OperatorUser:
    name: str | None = None
    tags: list[str] = ["ops"]

    def __init__(self, name: str) -> None:
        self.name = name

    def full_name(self) -> str:
        return f"{self.name}"


def print_name(obj: BaseUser) -> None:
    print(obj.full_name(), obj.tags)


ops_user = OperatorUser(name="Fra")
print_name(ops_user)
```

E se...

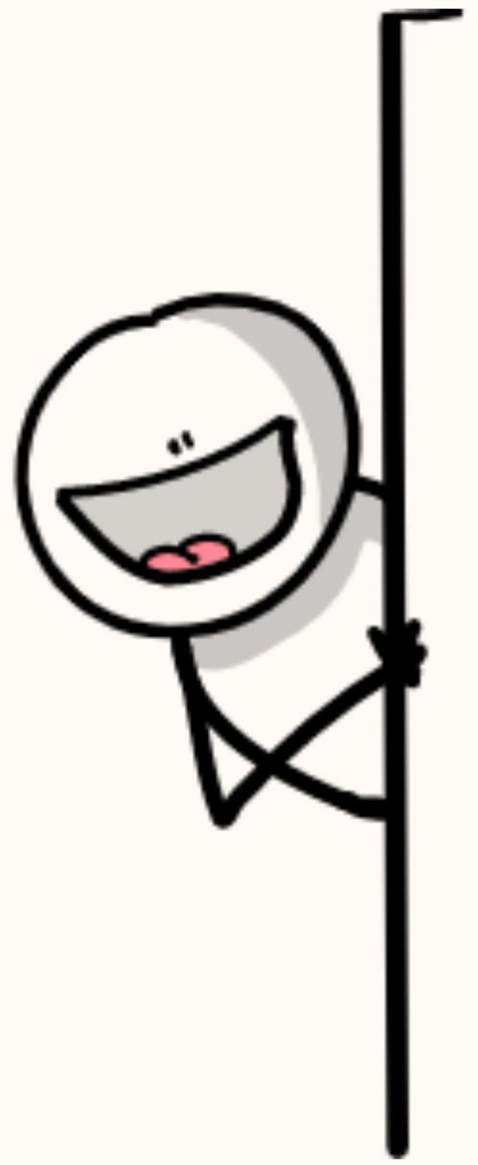
```python
class OperatorUser:
    name: str | None = None
    tags: list[str] = ["ops"]

    def __init__(self, name: str) -> None:
        self.name = name

    def full_name(self) -> str:
        return f"{self.name}"


def print_name(obj: BaseUser) -> None:
    print(obj.full_name(), obj.tags)


ops_user = OperatorUser(name="Fra")
print_name(ops_user)
# -> Fra ['ops']
```

E se...

```
❯ mypy 04.py
04.py:33: error: Argument 1 to "print_name" has incompatible type "OperatorUser";
expected "BusinessUser"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

Duck typing

```python
class Lion:
    def run(self) -> None:
        pass


class Gazelle:
    def run(self) -> None:
        pass


for animal in [Lion(), Gazelle()]:
    animal.run()
```



Ad ogni esecuzione, quando l'interprete si sveglia, non importa che tu sia un’istanza di `Lion()` o di `Gazelle()`… l’importante è che tu abbia il metodo `run()`

Structural Subtyping

```python
class A:
   def foo(self):
       pass


class B:  # <-- B does not extend A
   def foo(self):
       pass

   def bar(self):
       pass

   def baz(self):
       pass
```

Una classe *B* è un sottotipo di una classe *A* se i suoi metodi e le sue proprietà sono un sovrainsieme dei metodi e delle proprietà di *A*.

Lo structural subtyping è l'equivalente del duck typing applicato all'analisi statica.

L'introduzione dei Protocol (PEP 544)

Protocol: Structural Subtyping

```python
from typing import Protocol


class UserProtocol(Protocol):
    tags: list[str]

    def full_name(self) -> str: ...
```

```python
class OperatorUser:
    name: str | None = None
    tags: list[str] = ["ops"]

    def __init__(self, name: str) -> None:
        self.name = name

    def full_name(self) -> str:
        return f"{self.name}"
```

Protocol: Structural Subtyping

```python
class UserProtocol(Protocol):
    # cut


def print_name(obj: UserProtocol) -> None:
    print(obj.full_name(), obj.tags)


ops_user = OperatorUser(name="Fra")
print_name(ops_user)
```

Protocol: Structural Subtyping

```python
class UserProtocol(Protocol):
    # cut


def print_name(obj: UserProtocol) -> None:
    print(obj.full_name(), obj.tags)


ops_user = OperatorUser(name="Fra")
print_name(ops_user)
# --> Fra ['ops']
```

Protocol: Structural Subtyping

```
❯ mypy 08.py
Success: no issues found in 1 source file
```

Protocol: Structural Subtyping

```python
class FakeUser:
    pass


fake_user = FakeUser()
print_name(fake_user)
```

```
❯ mypy 08f.py
```

Protocol: Structural Subtyping

```python
class FakeUser:
    pass


fake_user = FakeUser()
print_name(fake_user)
# --> AttributeError: 'FakeUser' object has no attribute 'full_name'
```

```
❯ mypy 08f.py
08f.py:35: error: Argument 1 to "print_name" has incompatible type "FakeUser";
expected "UserProtocol"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

Obiettivo raggiunto

```python
from typing import Protocol


# - Interface
class UserProtocol(Protocol):
    tags: list[str]

    def full_name(self) -> str: ...


# - The function™
def print_name(obj: UserProtocol) -> None:
    print(obj.full_name(), obj.tags)
```

Obiettivo raggiunto

```python
# - Concrete class
class OperatorUser:
   name: str | None = None
   tags: list[str] = ["ops"]

   def __init__(self, name: str) -> None:
       self.name = name

   def full_name(self) -> str:
       return f"{self.name}"


class BusinessUser:
   first_name: str | None = None
   last_name: str | None = None
   tags: list[str]

   def __init__(self, first_name: str, last_name: str, tags: list[str]) -> None:
       self.first_name = first_name
       self.last_name = last_name
       self.tags = tags

   def full_name(self) -> str:
       return f"{self.first_name} - {self.last_name}"
```

Obiettivo raggiunto

```python
# - use case
ops_user = OperatorUser(name="Fra")
business_user = BusinessUser(first_name="BU", last_name="Bar",
tags=["red", "white"])

print_name(ops_user)


print_name(business_user)
```

Obiettivo raggiunto

```python
# - use case
ops_user = OperatorUser(name="Fra")
business_user = BusinessUser(first_name="BU", last_name="Bar",
tags=["red", "white"])

print_name(ops_user)
# --> Fra ['ops']

print_name(business_user)
# -> BU - Bar ['red', 'white']
```

```
❯ mypy 10.py
Success: no issues found in 1 source file
```

Checking a runtime: isinstance

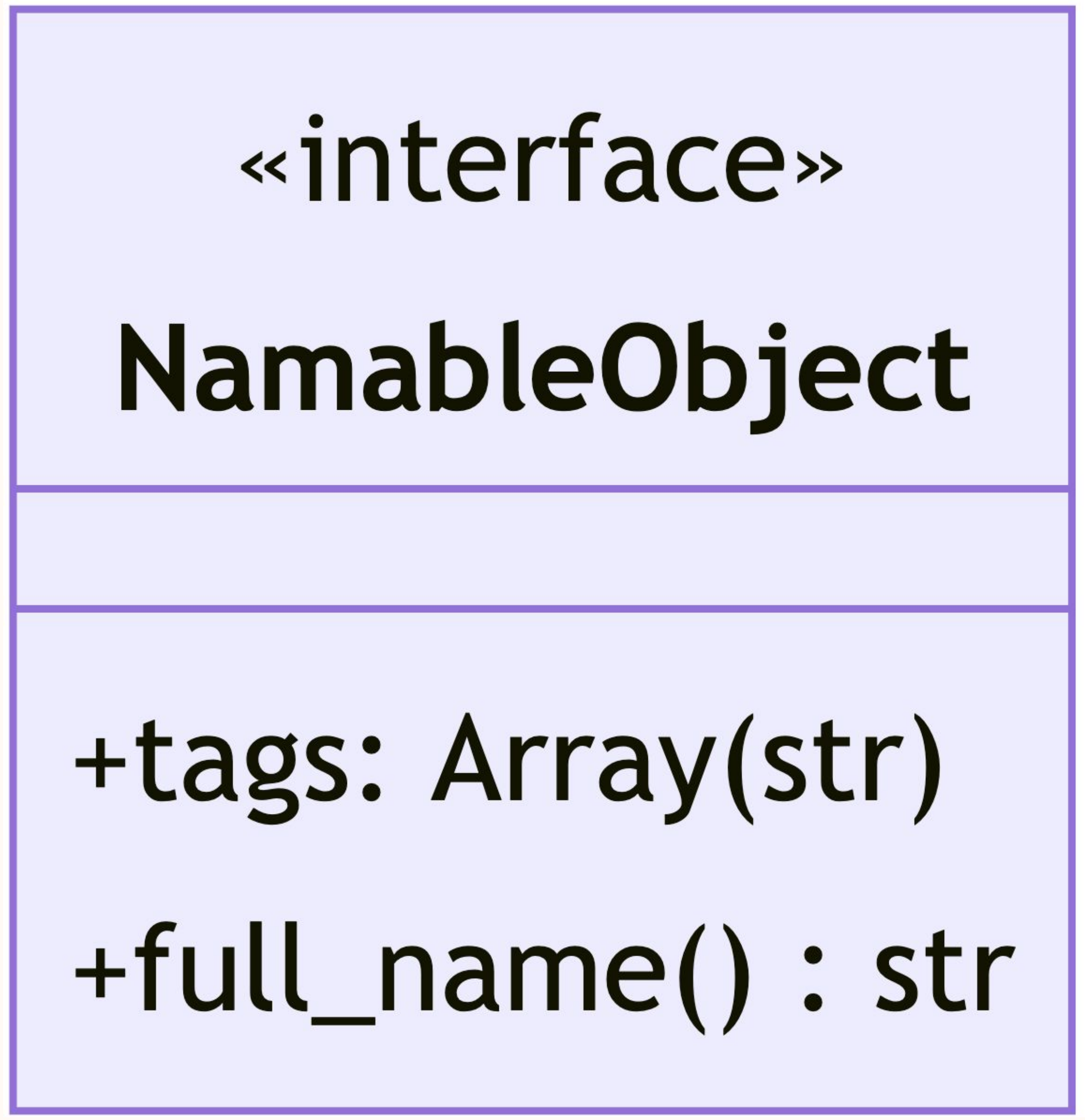
```python
import abc


class BaseUser(abc.ABC):
    @property
    @abc.abstractmethod
    def tags(self) -> list[str]:
        pass

    @abc.abstractmethod
    def full_name(self) -> str:
        pass


def print_name(obj: BaseUser) -> None:
    if isinstance(obj, BaseUser):
        print(obj.full_name(), obj.tags)
    else:
        print("wrong args")
```

```python
from typing import Protocol, runtime_checkable


@runtime_checkable
class UserProtocol(Protocol):
    tags: list[str]

    def full_name(self) -> str: ...


def print_name(obj: UserProtocol) -> None:
    if isinstance(obj, UserProtocol):
        print(obj.full_name(), obj.tags)
    else:
        print("wrong args")
```

Interface segregation

Interface segregation

```python
from typing import Protocol


# - Interface
class NamableObject(Protocol):
    tags: list[str]

    def full_name(self) -> str: ...


# - The function™
def print_name(obj: NamableObject) -> None:
    print(obj.full_name(), obj.tags)
```



Interface segregation

```python
from dataclasses import dataclass


# - Concrete class
@dataclass
class Peonia:
    tags: list[str]
    age: int
    name: str
    garden: Garden
    season: Season

    def full_name(self) -> str:
        return f"{self.name}"
```

```python
# - Concrete class
class BusinessUser(User):
    first_name: str | None = None
    last_name: str | None = None
    tags: list[str]

    def __init__(
        self, first_name: str,
        last_name: str,
        tags: list[str],
    ) -> None:
        self.first_name = first_name
        self.last_name = last_name
        self.tags = tags

    def full_name(self) -> str:
        return f"{self.first_name} - {self.last_name}"

    def logged(self) -> bool:
        return True
```

Interface segregation

```python
# - use case
business_user = BusinessUser(first_name="BU", last_name="Bar", tags=["red", "white"])
print_name(business_user)


plant = Peonia(
    name="Peonia in my garden",
    tags=["Pæonia", "Peoniacee"],
    age=1,
    garden=Garden(),
    season=Season(),
)
print_name(plant)
```

```
❯ mypy 11.py
```

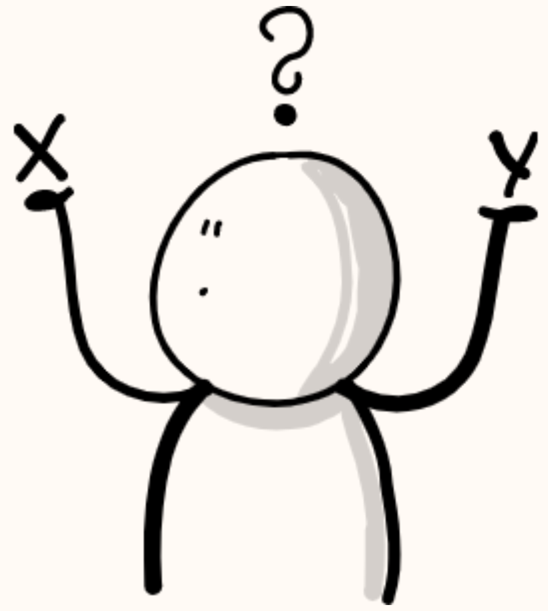
Interface segregation

```python
# - use case
business_user = BusinessUser(first_name="BU", last_name="Bar", tags=["red", "white"])
print_name(business_user)
# -> BU - Bar ['red', 'white']

plant = Peonia(
    name="Peonia in my garden",
    tags=["Pæonia", "Peoniacee"],
    age=1,
    garden=Garden(),
    season=Season(),
)
print_name(plant)
# -> Peonia in my garden ['Pæonia', 'Peoniacee']
```

```
❯ mypy 11.py
Success: no issues found in 1 source file
```

Nominal subtyping - Structural subtyping

```python
import abc


class BaseUser(abc.ABC):
   @property
   @abc.abstractmethod
   def tags(self) -> list[str]: pass

   @abc.abstractmethod
   def full_name(self) -> str: pass


class OperatorUser(BaseUser):
   name: str | None = None
   tags: list[str] = ["ops"]

   def __init__(self, name: str) -> None:
       self.name = name

   def full_name(self) -> str:
       return f"{self.name}"


def print_name(obj: BaseUser) -> None:
   print(obj.full_name(), obj.tags)
```




```python
from typing import Protocol


class UserProtocol(Protocol):
   tags: list[str]

   def full_name(self) -> str: ...


class OperatorUser:
   name: str | None = None
   tags: list[str] = ["ops"]

   def __init__(self, name: str) -> None:
       self.name = name

   def full_name(self) -> str:
       return f"{self.name}"


def print_name(obj: UserProtocol) -> None:
   print(obj.full_name(), obj.tags)
```

Utilizzo esplicito ed implicito dei Protocol

Quando una classe derivata include il Protocol nella lista delle sue classi base, si parla di utilizzo esplicito del Protocol (nominal subtyping).
Quando ne implementa proprietà e metodi senza utilizzare l'ereditarietà diretta, si parla al contrario di utilizzo implicito (structural subtyping).

L'utilizzo implicito è particolarmente interessante perché consente di definire interfacce anche per codice di terze parti, su cui non si ha un controllo diretto.

Domande?

Ringraziamenti

Fiscozen

PYCON(IT) 23