KRAKEN

Learning the ropes

Understanding Python Generics

David Seddon | Kraken Tech | Europython 2023

Do you use
a static type checker?

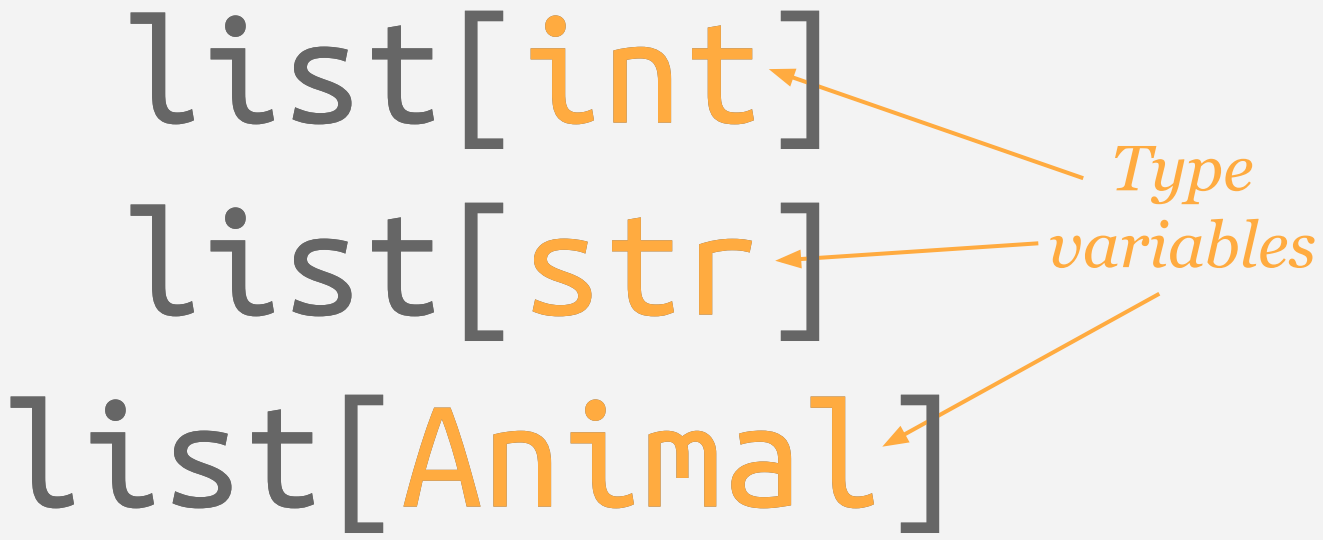

: my[py]

```
from typing import Generic
```

1.

Unexpected danger

You're already using
generics (probably).


list[int]
list[str]
list[Animal]
Type variables

```python
class Animal:
    def feed(self) -> None:
        print("Yum!")
```

```python
def feed_the_animals(animals: list[Animal]) -> None:
    for animal in animals:
        animal.feed()
```

```python
animals = [Animal(), Animal(), Animal()]
feed_the_animals(animals)
```

```
Yum!
Yum!
Yum!
```

```
Success: no issues found in 1 source file
```

```python
def feed_the_animals(animals: list[Animal]) -> None:
    for animal in animals:
        animal.feeed()
```

```
error: "Animal" has no attribute "feeed"; maybe "feed"?
```

```
class Cat(Animal):
   pass

feed_the_animals([Cat(), Cat(), Cat()])
```

```
Yum!
Yum!
Yum!
```

```
Argument 1 to "feed_the_animals" has
incompatible type "List[Cat]";
expected "List[Animal]"
```

2.

The Liskov Substitution Principle

An object may be replaced with a sub-object without breaking the program.

The Liskov Substitution Principle, paraphrased

An Animal may be replaced with a Cat without breaking the program.

So why is this a useful principle?

Polymorphism.

(Interacting with different types using the same interface.)

```
for animal in (Cat(), Dog()):
    animal.feed()
```

```
for animal in (Cat(), Dog()):
    if isinstance(animal, Cat):
        feed_cat(animal)
    elif isinstance(animal, Dog):
        feed_dog(animal)
```

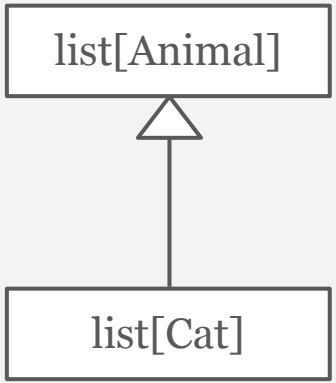
```
Argument 1 to "feed" has incompatible type
"List[Cat]"; expected "List[Animal]"
```

Why won't mypy let us interact with a `list[Cat]` as if it is a `list[Animal]`?

3.

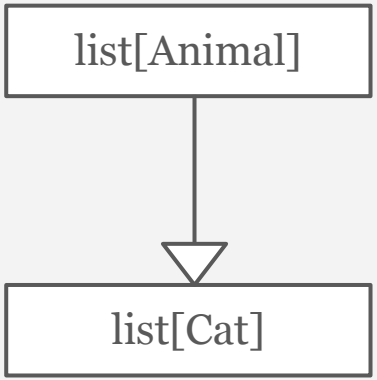
We need to talk about variance

Variance



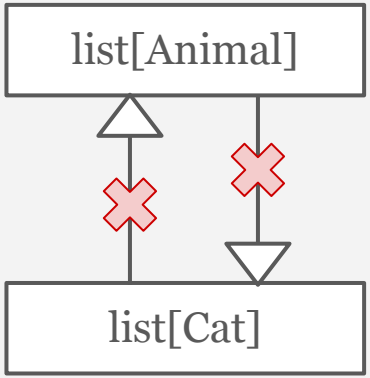
Covariant

or



Contravariant

or



Invariant

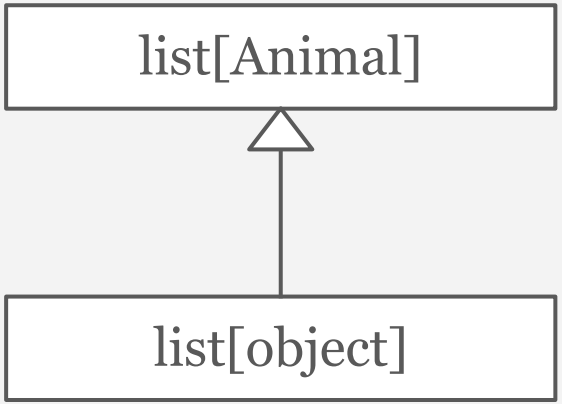
Variance is a design decision
for a typing system

Mypy's design decision
is to adhere to the
Liskov Substitution Principle

What should be the variance of `list`?

a. Covariant
b. Contravariant
c. Invariant

Could list be contravariant?




```
def feed_the_animals(animals: list[Animal]) -> None:
    for animal in animals:
        animal.feed()
```

```
objects = [object(), object(), object()]
feed_the_animals(objects)
```

```
AttributeError: 'object' object has no attribute 'feed'
```

```
error: Argument 1 to "feed_the_animals" has
incompatible type "List[object]"; expected
"List[Animal]"
```

Could list be covariant?

```
error: Argument 1 to "feed_the_animals" has
incompatible type "List[Cat]"; expected "List[Animal]"
```

```python
class Dog(Animal):
    pass


def increase(animals: list[Animal]) -> None:
    animals.append(Dog())
```

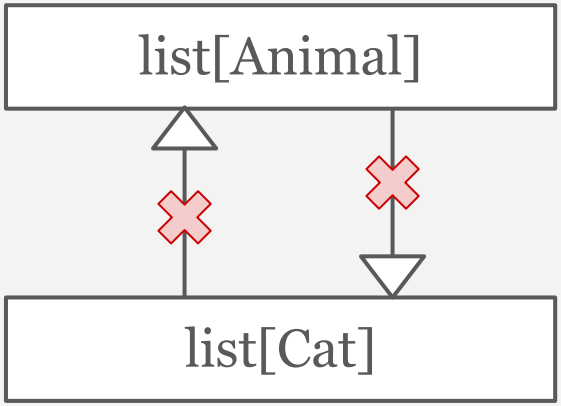
```python
cats = [Cat(), Cat(), Cat()]
increase(cats)
```

There is now a `Dog` in a `list[Cat]`!

```
animals: list[Animal] = [Cat(), Cat(), Cat()]
increase(animals)
```

```
Success: no issues found in 1 source file
```

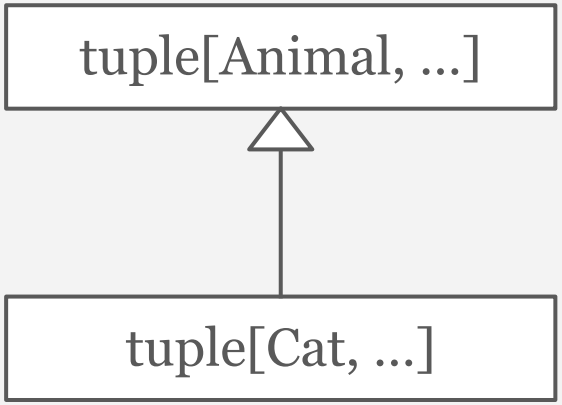
Lists are invariant.



This is because they're mutable collections.

tuple

Could tuple be covariant?




```
def feed_the_animals(animals: tuple[Animal, ...]) -> None:
    for animal in animals:
        animal.feed()
```

```
cats = (Cat(), Cat(), Cat())
feed_the_animals(cats)
```

```
Success: no issues found in 1 source file
```

Tuples and other immutable collections are covariant.

Variance of return types

```
class AnimalFinder:
    def find(self) -> Animal:
        ...
```

```
class CatFinder(AnimalFinder):
    def find(self) -> Cat:
        ...
```

covariant
(allowing this)...

```
class ObjectFinder(AnimalFinder):
    def find(self) -> object:
        ...
```

contravariant
(allowing this)...

... or invariant (allowing neither)?

```
class CatFinder(AnimalFinder):
    def find(self) -> Cat:
        ...
```

```
class ObjectFinder(AnimalFinder):
    def find(self) -> object:
        ...
```

```
finder: AnimalFinder = CatFinder()
animal: Animal = finder.find()
```

```
finder: AnimalFinder = ObjectFinder()
animal: Animal = finder.find()
```

Return types are covariant, too.

Variance of argument types

```python
class Animal:
    def feed(self, food: Food) -> None:
        ...
```

```python
class Cat(Animal):
    def feed(self, food: CatFood) -> None:
        ...
```

covariant
(allowing this)...

```python
class Cat(Animal):
    def feed(self, food: object) -> None:
        ...
```

contravariant
(allowing this)...

... or invariant (allowing neither)?

```python
class Cat(Animal):
  def feed(self, food: CatFood) -> None:
      ...
```

⚠️

```python
class Cat(Animal):
  def feed(self, food: object) -> None:
      ...
```

✅

```python
animal: Animal = Cat()
animal.feed(Food())
```

Argument types are contravariant.

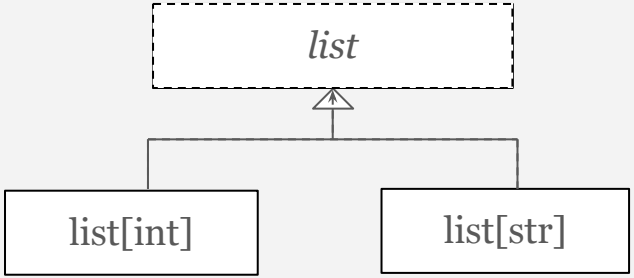
4.

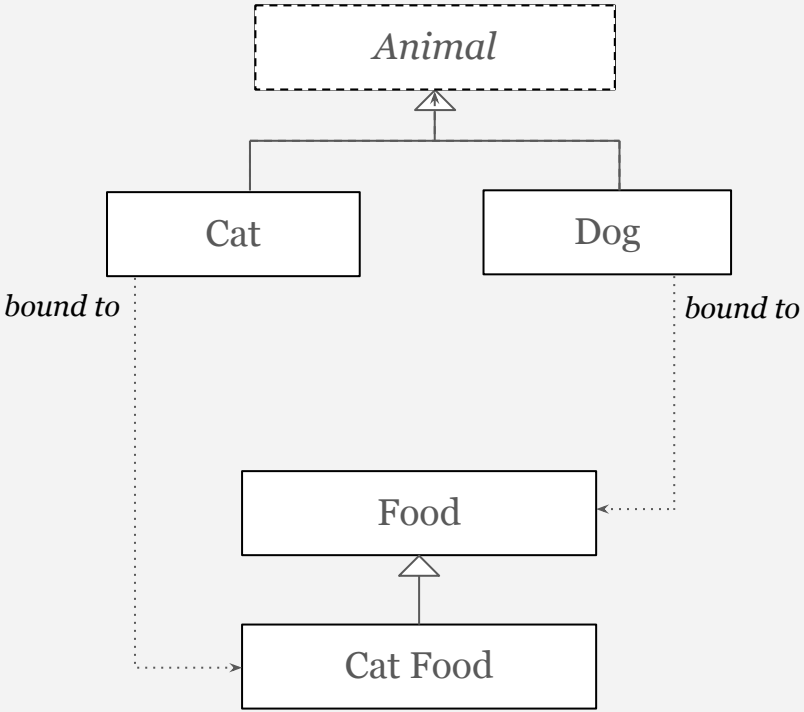
Custom generics

```
class Cat(Animal):
    def feed(self, food: CatFood) -> None:
        ...
```

What if we don't
want certain types
to be substitutable?

What if we don't want a
`Cat` to be an `Animal`?


list
list[int]
list[str]


Animal
Cat
Dog
bound to
bound to
Food
Cat Food

```python
from typing import TypeVar

T = TypeVar("T", bound=Food)
```

```python
from typing import Generic


class Animal(Generic[T]):
    ...
    def feed(self, food: T) -> None:
        ...
```

```
animal = Animal()
animal.feed(Food())
```

```
error: Need type annotation for "animal"
```

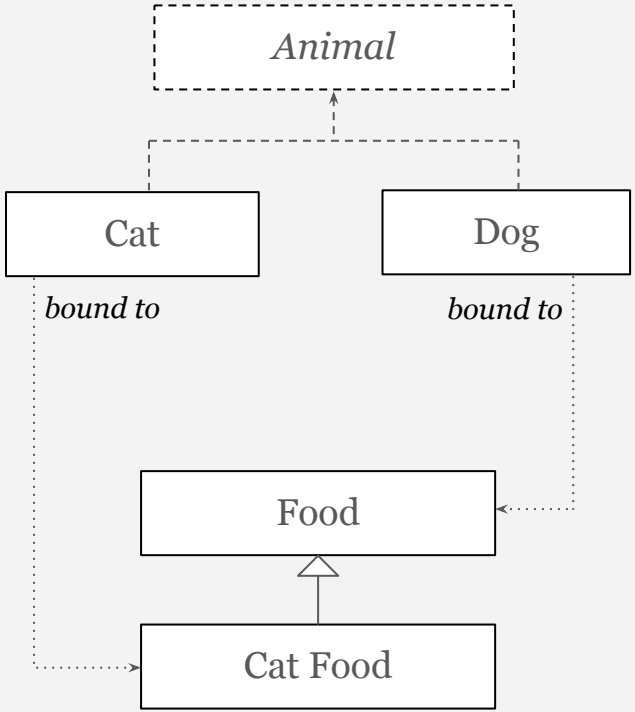
```
animal = Animal[CatFood]()
animal.feed(Food())
```

```
error: Argument 1 to "feed" of "Animal"
has incompatible type "Food";
expected "CatFood"
```

```
animal = Animal[CatFood]()
animal.feed(CatFood())
```

```
Success: no issues found in 1
source file
```

```
class Cat(Animal[CatFood]):
    ...
```



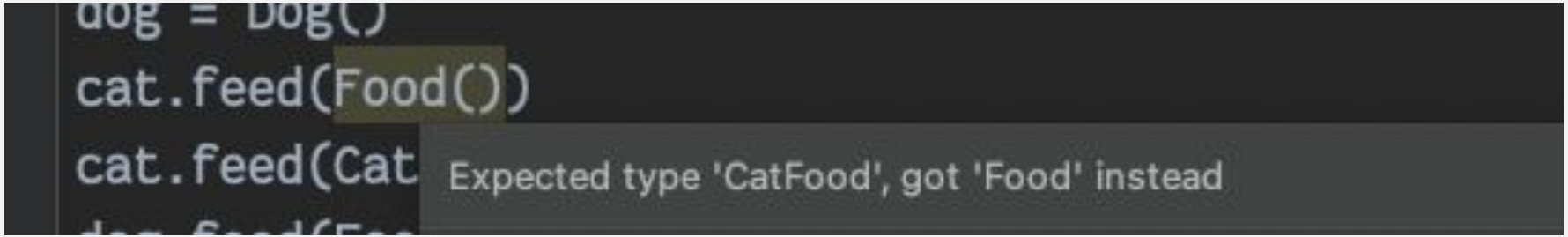

```
class Dog(Animal[Food]):
    ...
```

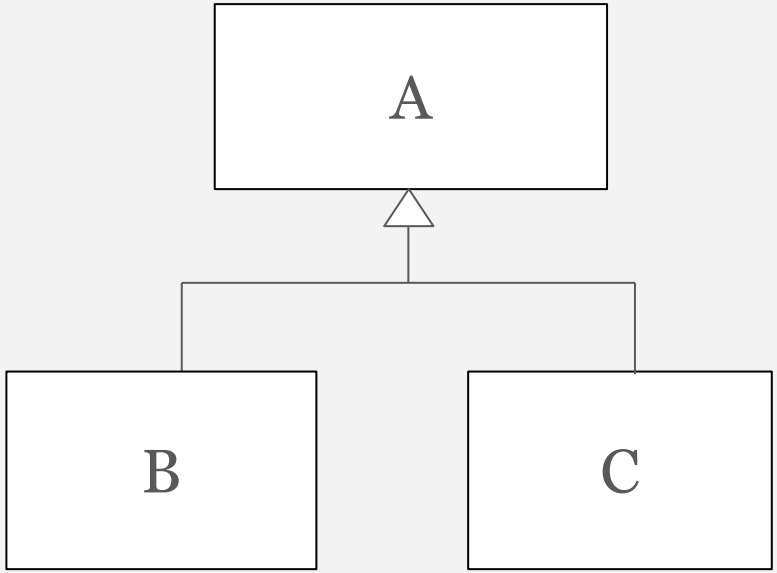
```python
class Cat(Animal[CatFood]):
    ...

    def feed(self, food: CatFood) -> bool:
        ...
```

```python
class Animal(Generic[T]):
    ...

    def feed(self, food: T) -> None:
        ...
```

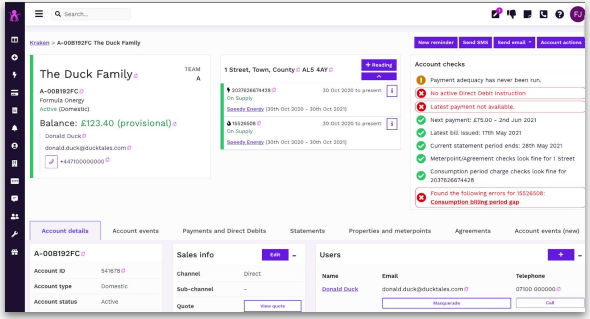

dog = Dog()
cat.feed(Food())
cat.feed(Cat
Expected type 'CatFood', got 'Food' instead


A
B
C

5.

Case study

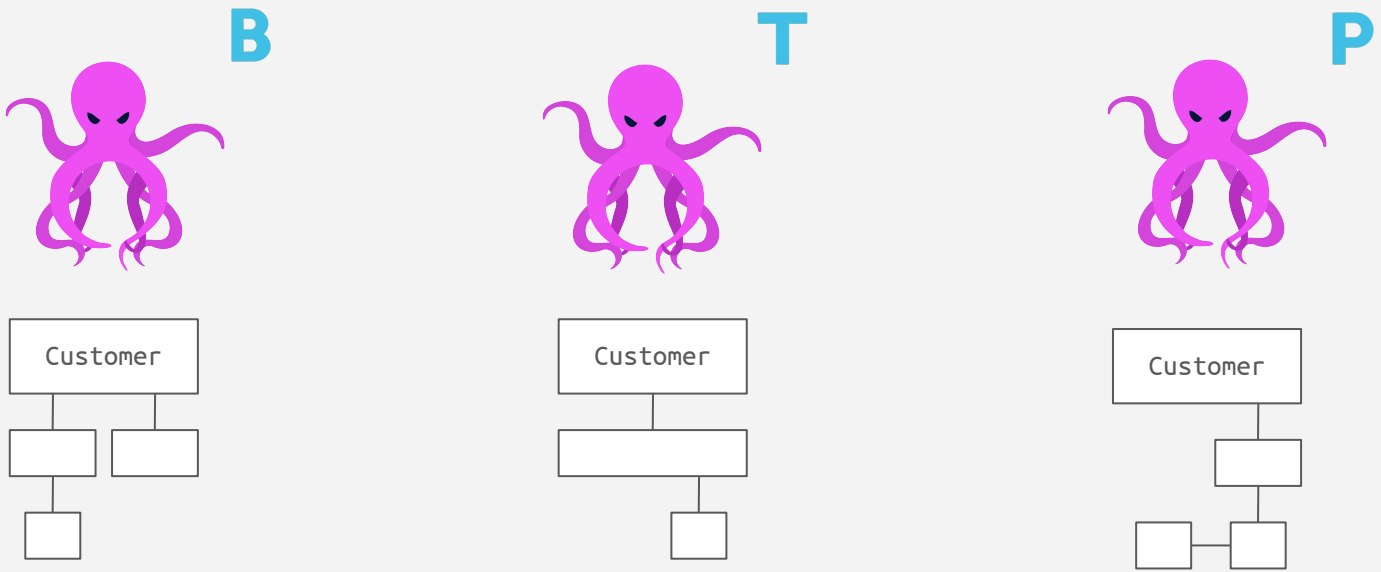
‘Kraken’



KRAKEN Technologies

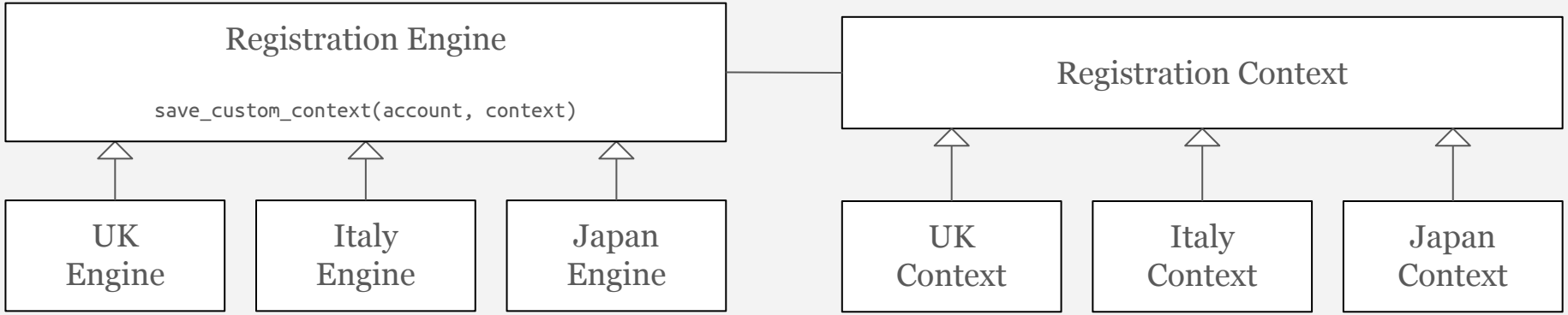
octopus energy

Customer registration in Kraken




```
def register(**kwargs: Any) -> Account:
    ...
```

Engines and contexts



```
register(
    engine=ItalyEngine(),
    context=ItalyContext(
        name="Marco Bianchi",
        email="marcob@gmail.com",
        address=Address(
            line1="Via delle Viole 11",
            line2="Povegliano Veronese",
            line3="Salerno",
            postal_code="14026",
        ),
        phone_number="0397 8142253",
        opted_in_to_marketing=True,
        fiscal_code="MRTMTT91D08F205J",
    ),
)
```

```
def register(engine: RegistrationEngine, context: RegistrationContext) -> Account:
    account = _create_account(context)

    engine.save_custom_context(account, context)

    return account
```

```
class RegistrationEngine:
    def save_custom_context(
        self, account: Account, context: RegistrationContext
    ) -> None:
        pass
```

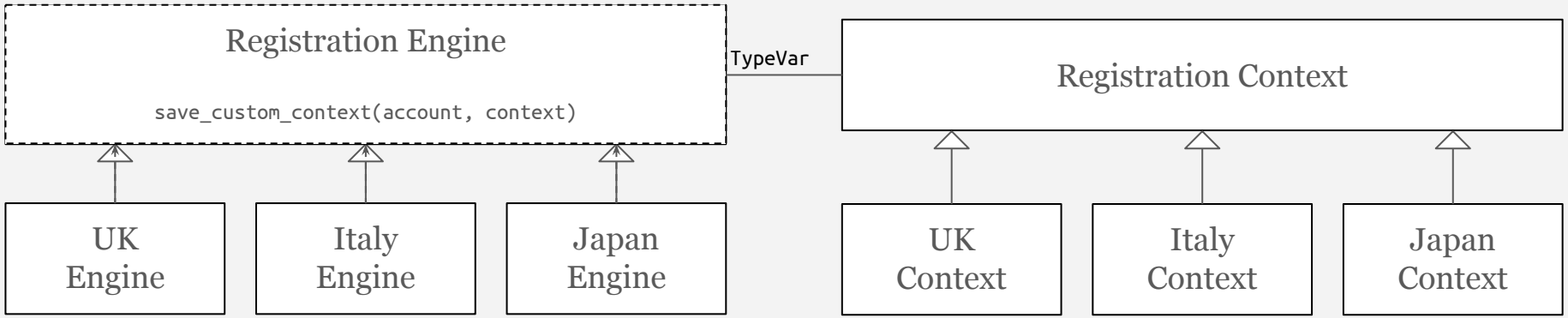
```python
@dataclass
class RegistrationContext:
    name: str
    email: str
    address: Address
    phone_number: str
    opted_in_to_marketing: bool
```

```python
@dataclass
class ItalyContext(RegistrationContext):
    fiscal_code: str


class ItalyEngine(RegistrationEngine):
    def save_custom_context(self, account: Account, context: ItalyContext) -> None:
        save_italy_data(account=account, fiscal_code=context.fiscal_code)
```

```
error: Argument 2 of "save_custom_context" is incompatible with supertype
"RegistrationEngine"; supertype defines the argument type as "RegistrationContext"
note: This violates the Liskov substitution principle
```

So what's the answer?


Registration Engine
save_custom_context(account, context)
TypeVar
Registration Context
UK Engine
Italy Engine
Japan Engine
UK Context
Italy Context
Japan Context

```python
from typing import Generic, TypeVar

T = TypeVar("T", bound=RegistrationContext)


class RegistrationEngine(Generic[T]):
    def save_custom_context(self, account: Account, context: T) -> None:
        pass
```

```python
class ItalyEngine(RegistrationEngine[ItalyContext]):
    def save_custom_context(self, account: Account, context: ItalyContext) -> None:
        ...
```

```python
def register(engine: RegistrationEngine[T], context: T) -> Account:
    ...
```

```
Success: no issues found in 1 source file 😀 😀 😀
```

Thank you!

seddonym.me

import-linter.readthedocs.io

We're hiring: octopus.energy/kraken-tech-careers

Appendix

```
dict[str, Animal]
```

type of the keys

type of the values

Summary of variance (as Mypy sees it)

Mutable collections (`list, set`)	Invariant
Immutable collections (`tuple, frozenset, Sequence`)	Covariant
Return types	Covariant
Argument types	Contravariant

```python
def feed_the_animals(animals: list[Animal]) -> None:
    for animal in animals:
        animal.feeed()
```

```
error: "Animal" has no attribute "feeed"; maybe "feed"?
```

```python
feed_the_animals([Animal(), 33, Animal()])
```

```
error: List item 1 has incompatible type "int"; expected "Animal"
```