Annotated, a “type hint” you can use at runtime

Denis Laxalde

February, 4th 2024, FOSDEM

`Annotated`: a *type hint* you can use at **runtime**

About me

- programming with Python for ~15 years for fun and profit
- work at Dalibo[1], PostgreSQL services in France
  - developing products for the database infrastructure automation
  - contributing to the PostgreSQL ecosystem
- contributor to:
  - *Psycopg*, *pg_activity*
  - and less recently: Mercurial, Scipy, CubicWeb
- @dlax / denis@laxalde.org[2]

[1] https://dalibo.com/en/
[2] mailto:denis@laxalde.org

Annotated: why?

```
class User(BaseModel):
    id: Annotated[str, Field(default_factory=lambda: uuid4().hex)]
    name: str
```

Annotated: why?

```python
class User(BaseModel):
    id: Annotated[str, Field(default_factory=lambda: uuid4().hex)]
    name: str
```

```python
@app.get("/users")
def users(q: Annotated[str | None, Query(max_length=50)] = None) -> list[User]:
    ...
```

Annotated: why?

```python
class User(BaseModel):
    id: Annotated[str, Field(default_factory=lambda: uuid4().hex)]
    name: str
```

```python
@app.get("/users")
def users(q: Annotated[str | None, Query(max_length=50)] = None) -> list[User]:
    ...
```

Motivation

- How does it work?

Annotated: why?

```python
class User(BaseModel):
    id: Annotated[str, Field(default_factory=lambda: uuid4().hex)]
    name: str
```

```python
@app.get("/users")
def users(q: Annotated[str | None, Query(max_length=50)] = None) -> list[User]:
    ...
```

Motivation

- How does it work?
- How do I define `MyAnnotation` in `Annotated[str, MyAnnotation]`?

Annotated: why?

```python
class User(BaseModel):
    id: Annotated[str, Field(default_factory=lambda: uuid4().hex)]
    name: str
```

```python
@app.get("/users")
def users(q: Annotated[str | None, Query(max_length=50)] = None) -> list[User]:
    ...
```

Motivation

- How does it work?
- How do I define `MyAnnotation` in `Annotated[str, MyAnnotation]`?
- For which *use cases*?

Outline

1. Introducing `typing.Annotated` and PEP-593
2. Use cases: data models, validation, serialization, UI
3. Adoption in the community and ecosystem

PEP 593

`from typing import Annotated`

- not really a “type hint”

[3] https://peps.python.org/pep-3107/
[4] https://peps.python.org/pep-484/
[5] https://pypi.org/project/typing-extensions/
[6] https://peps.python.org/pep-0593/

`from typing import Annotated`

- not really a “type hint”
- rather an *annotation* (`<identifier>: <annotation>`)

  . . . maybe more in the spirit of PEP 3107 — Function Annotations[3] than from PEP 484 – Type Hints[4]

[3] https://peps.python.org/pep-3107/
[4] https://peps.python.org/pep-484/
[5] https://pypi.org/project/typing-extensions/
[6] https://peps.python.org/pep-0593/

`from typing import Annotated`

- not really a “type hint”
- rather an *annotation* (`<identifier>: <annotation>`)

  . . . maybe more in the spirit of PEP 3107 — Function Annotations[3] than from PEP 484 – Type Hints[4]
- from Python 3.9, or in typing-extensions[5]

[3] https://peps.python.org/pep-3107/
[4] https://peps.python.org/pep-484/
[5] https://pypi.org/project/typing-extensions/
[6] https://peps.python.org/pep-0593/

`from typing import Annotated`

- not really a “type hint”
- rather an *annotation* (`<identifier>: <annotation>`)

  . . . maybe more in the spirit of PEP 3107 — Function Annotations[3] than from PEP 484 – Type Hints[4]
- from Python 3.9, or in typing-extensions[5]
- PEP 593 – Flexible function and variable annotations[6]

[3] https://peps.python.org/pep-3107/
[4] https://peps.python.org/pep-484/
[5] https://pypi.org/project/typing-extensions/
[6] https://peps.python.org/pep-0593/

PEP 593 – Flexible function and variable annotations

```
v: Annotated[T, *x]
```

- `v`: a "name" (variable, function parameter, ...)
- `T`: a valid type
- `x`: at least one metadata (or annotation), passed in a variadic way
  The metadata can be used for either static analysis or at runtime.

PEP 593 – Flexible function and variable annotations

`v: Annotated[T, *x]`

- `v`: a "name" (variable, function parameter, ...)
- `T`: a valid type
- `x`: at least one metadata (or annotation), passed in a variadic way
  The metadata can be used for either static analysis or at runtime.

Composable
: *When a tool or a library does not support annotations or encounters an unknown annotation it should just ignore it and treat annotated type as the underlying type.*

Consuming annotations, getting “type hints”

Get type hints for an object, a class or a function

```
from typing import get_type_hints
```

Consuming annotations, getting “type hints”

Get type hints for an object, a class or a function

```
from typing import get_type_hints
```

```
@dataclass
class Point:
   x: int
   y: Annotated[int, Label("ordinate")]
```

```
>>> hints = get_type_hints(Point, include_extras=True)
>>> hints
{'x': <class 'int'>, 'y': typing.Annotated[int, Label('ordinate')]}
```

Consuming annotations, getting “type hints”

Get type hints for an object, a class or a function

```
from typing import get_type_hints
```

```
@dataclass
class Point:
   x: int
   y: Annotated[int, Label("ordinate")]
```

```
>>> hints = get_type_hints(Point, include_extras=True)
>>> hints
{'x': <class 'int'>, 'y': typing.Annotated[int, Label('ordinate')]}
```

`obj.__annotations__` may also be used but `get_type_hints()` can handle *forward references*

Consuming annotations, getting them from “type hints”

```
>>> hints
{'x': <class 'int'>, 'y': typing.Annotated[int, Label('ordinate')]}
```

Consuming annotations, getting them from “type hints”

```
>>> hints
{'x': <class 'int'>, 'y': typing.Annotated[int, Label('ordinate')]}
```

Inspect individual annotations

```
from typing import get_origin, get_args
```

Consuming annotations, getting them from “type hints”

```
>>> hints
{'x': <class 'int'>, 'y': typing.Annotated[int, Label('ordinate')]}
```

Inspect individual annotations

```
from typing import get_origin, get_args
```

```
>>> typing.get_origin(hints['y'])
<class 'typing.Annotated'>
```

Consuming annotations, getting them from “type hints”

```
>>> hints
{'x': <class 'int'>, 'y': typing.Annotated[int, Label('ordinate')]}
```

Inspect individual annotations

```
from typing import get_origin, get_args
```

```
>>> typing.get_origin(hints['y'])
<class 'typing.Annotated'>
```

```
>>> y_type, *y_annotations = typing.get_args(hints['y'])
>>> y_type, y_annotations
(<class 'int'>, [Label(name='ordinate')])
```

Consuming annotations, handling annotated values

```
>>> y_type, y_annotations
(<class 'int'>, [Label(name='ordinate')])
```

Handle *your* annotations (and ignore others')

```
>>> for a in y_annotations:
...     if not isinstance(a, Label):
...         continue
...     ...
```

Consuming annotations, the `get_annotations()` helper (simplistic)

```
from typing import Annotated, get_args, get_origin, get_type_hints

A = TypeVar("A")

def get_annotations(
    obj: object, atype: type[A]
) -> Iterator[tuple[str, A, type]]:
    """Yield annotations of specified type from 'obj'."""
    for key, hints in get_type_hints(obj, include_extras=True).items():
        if get_origin(hints) is Annotated:
            tp, *annotations = get_args(hints)
            for a in annotations:
                if isinstance(a, atype):
                    yield key, a, tp
```

```
>>> list(get_annotations(Point, Label))
[('y', Label('ordinate'), <class 'int'>)]
```

Use cases

A calendar `Event` model, using pydantic[7]

```python
from pydantic import BaseModel

class Event(BaseModel):
    summary: str
    description: str | None = None
    start_at: datetime | None = None
    end_at: datetime | None = None
```

[7] https://github.com/pydantic/pydantic

A calendar `Event` model, using pydantic[7]

```python
from pydantic import BaseModel

class Event(BaseModel):
    summary: str
    description: str | None = None
    start_at: datetime | None = None
    end_at: datetime | None = None
```

Next, let's add:

1. validation on `datetime` fields (using Pydantic)
2. *iCalendar* serialization support
3. console rendering

[7] https://github.com/pydantic/pydantic

datetime validation, validator annotations

```
from pydantic import AfterValidator

class Event(BaseModel):
    ...
    start_at: Annotated[datetime | None, AfterValidator(tz_aware)] = None
    end_at: Annotated[datetime | None, AfterValidator(tz_aware)] = None
```

datetime validation, validator annotations

```python
from pydantic import AfterValidator

class Event(BaseModel):
    ...
    start_at: Annotated[datetime | None, AfterValidator(tz_aware)] = None
    end_at: Annotated[datetime | None, AfterValidator(tz_aware)] = None
```

```python
def tz_aware(d: datetime) -> datetime:
    if d.tzinfo is None or d.tzinfo.utcoffset(d) is None:
        raise ValueError("expecting a TZ-aware datetime")
    return d
```

datetime validation, illustrated

```
>>> Event(summary="fosdem", start_at="2024-02-03T09:00:00")
Traceback (most recent call last):
  ...
pydantic_core....ValidationError: 1 validation error for Event
start_at
  Value error, expecting a TZ-aware datetime [
    ..., input_value='2024-02-03T09:00:00', ...
  ]
    ...
```

Side step: Pydantic validation, without `Annotated`

```python
from pydantic import field_validator

class Event(BaseModel):
    ...
    start_at: datetime | None = None
    end_at: datetime | None = None

    @field_validator("start_at", "end_at")
    @classmethod
    def validate_tz_aware(cls, value: datetime | None) -> datetime | None:
        return tz_aware(value) if value is not None else None
```

Side step: Pydantic validation, without `Annotated`

```python
from pydantic import field_validator

class Event(BaseModel):
    ...
    start_at: datetime | None = None
    end_at: datetime | None = None

    @field_validator("start_at", "end_at")
    @classmethod
    def validate_tz_aware(cls, value: datetime | None) -> datetime | None:
        return tz_aware(value) if value is not None else None
```

(Arguably) less convenient because:

- the validation method is loosely bound to attributes
- the method must be repeated for all model classes
- and similarly for serializers

Annotation alias, working around verbosity

```
TZDatetime = Annotated[datetime, AfterValidator(tz_aware)]

class Event(BaseModel):
    summary: str
    description: str | None
    start_at: TZDatetime | None
    end_at: TZDatetime | None
```

iCalendar serialization, annotating fields

```python
from . import ical

class Event(BaseModel):
    summary: Annotated[str, ical.Serializer(label="summary")]
    description: Annotated[str | None, ical.Serializer(label="description")] = None
    start_at: Annotated[TZDatetime | None, ical.Serializer(label="dtstart")] = None
    end_at: Annotated[TZDatetime | None, ical.Serializer(label="dtend")] = None
```

iCalendar annotation types and serialization logic

```python
# module: ical

@dataclass
class Serializer:
    label: str

    def serialize(self, value: Any) -> str:
        if isinstance(value, datetime):
            value = value.astimezone(timezone.utc).strftime("%Y%m%dT%H%M%SZ")
        return f"{self.label.upper()}:{value}"
```

iCalendar annotation types and serialization logic

```python
# module: ical

@dataclass
class Serializer:
    label: str

    def serialize(self, value: Any) -> str:
        if isinstance(value, datetime):
            value = value.astimezone(timezone.utc).strftime("%Y%m%dT%H%M%SZ")
        return f"{self.label.upper()}:{value}"
```

```python
def serialize_event(obj: Event) -> str:
    lines = []
    for name, a, _ in get_annotations(obj, Serializer):
        if (value := getattr(obj, name, None)) is not None:
            lines.append(a.serialize(value))
    return "\n".join(["BEGIN:VEVENT"] + lines + ["END:VEVENT"])
```

iCalendar serialization, illustrated

```
>>> evt = Event(
...     summary="FOSDEM",
...     start_at=datetime(2024, 2, 3, 9, 00, 0, tzinfo=ZoneInfo("Europe/Brussels")),
...     end_at=datetime(2024, 2, 4, 17, 00, 0, tzinfo=ZoneInfo("Europe/Brussels")),
... )
>>> print(ical.serialize_event(evt))
BEGIN:VEVENT
SUMMARY:FOSDEM
DTSTART:20240203T080000Z
DTEND:20240204T160000Z
END:VEVENT
```

Wrap up: defining and consuming custom annotations

1. define annotation types

```
class MyAnnotation:
    option: ...
    def handle(self, value: V, tp: type[V]): ...
```

Wrap up: defining and consuming custom annotations

1. define annotation types

```
class MyAnnotation:
    option: ...
    def handle(self, value: V, tp: type[V]): ...
```

2. annotate data structure fields, function parameters

```
x: Annotated[<type>, MyAnnotation(option=...), ...]
```

Wrap up: defining and consuming custom annotations

1. define annotation types

```
class MyAnnotation:
    option: ...
    def handle(self, value: V, tp: type[V]): ...
```

2. annotate data structure fields, function parameters

```
x: Annotated[<type>, MyAnnotation(option=...), ...]
```

3. consume objects' annotations at runtime

```
for name, a, tp in get_annotations(x, MyAnnotation):
    value = getattr(obj, name)
    a.handle(value, tp)
```

User interface, annotating fields

```
from . import ui

class Event(BaseModel, ui.Renderable):
    summary: Annotated[
        str,
        ical.Serializer(label="summary"),
        ui.Text(style="magenta bold"),
    ]
    description: Annotated[str | None, ui.Markdown()] = None
    start_at: Annotated[
        TZDatetime | None,
        ical.DateSerializer(label="dtstart"),
        ui.DateRelative(label="starts", style="green"),
    ] = None
    ...
```

Using rich[8], *a formatting library in the terminal*, to define UI widgets.

[8] https://github.com/Textualize/rich

UI widgets (annotation types)

```python
# module: ui
class Widget(ABC):
    @abstractmethod
    def render(self, value: Any) -> rich.abc.Renderable: ...

class Markdown(Widget):
    def render(self, value: str) -> rich.markdown.Markdown:
        return rich.markdown.Markdown(value, **self.options)

@dataclass
class DateRelative(Widget):
    label: str
    style: str | Style = ""

    def render(self, value: datetime) -> rich.text.Text:
        """Render date value with specified style if after current date."""
        ...
```

Consuming `ui` annotations and console rendering

Rich[9] *supports a simple protocol to add rich formatting capabilities to custom objects.*

```
# module: ui

class Renderable:
    def __rich_console__(self, *args):
        for name, a, _ in get_annotations(self, Widget):
            if (rendered := a.render(getattr(self, name, None))) is not None:
                yield rendered
```

used as a *mixin*:

```
class Event(BaseModel, ui.Renderable):
    ...
```

[9] https://github.com/Textualize/rich

Console rendering, illustrated

```
>>> evt = Event(
...     summary="FOSDEM'24",
...     description="""\
... ## What is FOSDEM?
... FOSDEM is a free and non-commercial event organised by the community for
... the community. The goal is to ...
...
... -   get in touch with other developers and projects;
... -   ...
... """,
...     start_at=datetime(2024, 2, 3, 9, tzinfo=ZoneInfo("Europe/Brussels")),
...     end_at=datetime(2024, 2, 4, 17, tzinfo=ZoneInfo("Europe/Brussels"))),
... )
```

Console rendering, illustrated

```
>>> evt = Event(
...     summary="FOSDEM'24",
...     description="""\
... ## What is FOSDEM?
... FOSDEM is a free and non-commercial event organised by the community for
... the community. The goal is to ...
...
... -   get in touch with other developers and projects;
... -   ...
... """,
...     start_at=datetime(2024, 2, 3, 9, tzinfo=ZoneInfo("Europe/Brussels")),
...     end_at=datetime(2024, 2, 4, 17, tzinfo=ZoneInfo("Europe/Brussels"))),
... )
```

```
>>> rich.print(evt)
```

FOSDEM'24

What is FOSDEM?

FOSDEM is a free and non-commercial event organised by the community for the community. The goal is to ...

- get in touch with other developers and projects;
- ...

Starts: 2024-02-03 09:00:00
Ends: 2024-02-04 17:00:00

`Annotated` in the ecosystem and community

Adopters

[10] https://github.com/annotated-types/annotated-types

Adopters

Pydantic, FastAPI, Typer...

```python
@app.get("/events/")
async def get_events(
    q: Annotated[str | None, fastapi.Query(title="Search terms")] = None,
    user_agent: Annotated[str | None, fastapi.Header()] = None,
):
    ...
```

See also annotated-types[10], a library of reusable constraint types to use with `Annotated`.

[10] https://github.com/annotated-types/annotated-types

Adopters

Pydantic, FastAPI, Typer...

```python
@app.get("/events/")
async def get_events(
    q: Annotated[str | None, fastapi.Query(title="Search terms")] = None,
    user_agent: Annotated[str | None, fastapi.Header()] = None,
):
    ...
```

See also annotated-types[10], a library of reusable constraint types to use with `Annotated`.

SQLAlchemy

```python
class Event(DeclarativeBase):
    id: Mapped[Annotated[int, mapped_column(primary_key=True)]]
```

[10] https://github.com/annotated-types/annotated-types

Adopters

Pydantic, FastAPI, Typer...

```python
@app.get("/events/")
async def get_events(
    q: Annotated[str | None, fastapi.Query(title="Search terms")] = None,
    user_agent: Annotated[str | None, fastapi.Header()] = None,
):
    ...
```

See also annotated-types[10], a library of reusable constraint types to use with `Annotated`.

SQLAlchemy

```python
class Event(DeclarativeBase):
    id: Mapped[Annotated[int, mapped_column(primary_key=True)]]
```

Lesser enthusiam in projects with less coupling with the typing system...

[10] https://github.com/annotated-types/annotated-types

Scepticism. . .

- `Annotated` is `...` verbose

[11] https://peps.python.org/pep-0727/

Scepticism. . .

- `Annotated` is `...` verbose
- Annotations are not (necessarily) typing (though most consumers do use the typing information)

[11] https://peps.python.org/pep-0727/

Scepticism...

- `Annotated` is ... verbose
- Annotations are not (necessarily) typing (though most consumers do use the typing information)
  - How to use annotations for non-typed objects?

[11] https://peps.python.org/pep-0727/

Scepticism. . .

- `Annotated` is . . . verbose
- Annotations are not (necessarily) typing (though most consumers do use the typing information)
  - How to use annotations for non-typed objects?
- Consuming annotations (`typing.get_type_hints()`/ `typing.get_args()`, esp. for "special forms") can be tedious/fragile

[11] https://peps.python.org/pep-0727/

Scepticism. . .

- `Annotated` is . . . verbose
- Annotations are not (necessarily) typing (though most consumers do use the typing information)
  - How to use annotations for non-typed objects?
- Consuming annotations (`typing.get_type_hints()`/ `typing.get_args()`, esp. for "special forms") can be tedious/fragile
- More non-typing *metadata* coming, e.g., in PEP 727: Documentation Metadata in Typing[11]:

```
from typing import Annotated, Doc

class User:
    name: Annotated[str, Doc("The user's name")]
```

[11] https://peps.python.org/pep-0727/

And beyond, the “typing” topic

- Quite “divisive” in the community

[12] https://lwn.net/Articles/958326/

And beyond, the “typing” topic

- Quite “divisive” in the community
- Python is growing with these features, bringing *user value*

[12] https://lwn.net/Articles/958326/

And beyond, the “typing” topic

- Quite “divisive” in the community
- Python is growing with these features, bringing *user value*
- Further reading: Growing pains for typing in Python[12]

[12] https://lwn.net/Articles/958326/

Thank you!