Published in The Startup

Gideon Caller Follow

Oct 6, 2020 · 11 min read · Listen

Cool Things You Can Do With Pydantic

```python
from typing import Union
from typing_extensions import Literal
from pydantic import BaseModel, conlist, constr, ValidationError, BaseSettings


class DevContext(BaseSettings):
    env: Literal["dev"]
    local_value: int


class ProdContext(BaseSettings):
    env: Literal["prod"]
    url: str


class Context(BaseSettings):
    __root__: Union[DevContext, ProdContext]

    @classmethod
    def create(cls):
        model = cls.parse_obj({})
        return model.__root__


class Age(BaseModel):
    __root__: int


class Pizza(BaseModel):
    toppings_count: int
    size: str


class User(BaseModel):
    name: constr(min_length=1)
    scores: conlist(int, min_items=1)

```

Pydantic models

Pydantic is a useful library for data parsing and validation. It coerces input types to the declared type (using type hints), accumulates all the errors using `ValidationError` & it's also well documented making it easily discoverable.

During my time using Pydantic, I picked up a few things that were not immediately obvious to me and also bumped into a few pitfalls. Since it took me a while to discover these, I figured it's time to share them with the world.

Before we begin a few things to note:

- I've added numbered comments in the code (# 1, # 2, etc) which I immediately refer to after the code snippet in order to explain the code.
- Each feature/pitfall has a link in the following section so you jump directly to the ones that interest you.

So without further ado, here are the things I learned you can do with Pydantic:

Use field aliases to play nicely with external formats

When data passes through our system boundaries like external APIs, DBs, messaging queues, etc we sometimes need to follow others' naming conventions (CamelCase vs snake_case, etc).

In some cases, it may lead to some weird Python conventions:

```python
from pydantic import BaseModel


class Item(BaseModel):
    id: str  # 1
    isAvailable: bool  # 2


def test_does_it_look_pythonic_to_you():
    item = Item(id='test-item-id', isAvailable=True)  # 3
    assert item.isAvailable is True  # 3

    external_input = {'id': 'new-test-id', 'isAvailable': False}
    item = Item(**external_input)  # 4
    assert item.isAvailable is False  # 5
```

camelcase_naming_example.py hosted with ❤ by **GitHub** **view raw**

1. ~~"id" is a reserved keyword in Python~~ `id` is a built-in function in Python (Jylpah correctly pointed out that `id` is a function and not a reserved keyword in Python as I initially wrote) but we must use it in order to follow an external naming convention.
2. In Python, we generally use snake_case as the naming convention but again, we are forced to follow a different naming convention.
3. Since we want a model to represent the external data we are forced to follow different conventions.
4. We may also receive data from an external source in the form of JSON or some other format.
5. We are still forced to follow these external conventions.

In other cases, it may yield surprising results:

1. In order to avoid using `id` as the field name (as it's a reserved keyword), we rename our field.
2. Surprisingly (or at least surprising to me), Pydantic hides fields that start with an underscore (regardless of how you try to access them).

Pydantic allows us to overcome these issues with field aliases:

1. This is how we declare a field alias in Pydantic. Note how the alias should match the external naming conventions.
2. When creating models with aliases we pass inputs that match the aliases.
3. We access the field via the field name (and not the field alias).
4. Beware of trying to create these models with the actual field names (and not the aliases) as it will not work. Check out the attached Github issue to learn more about this.

5. When converting our models to external formats we need to tell Pydantic to use the alias (instead of the internal name) using the `by_alias` argument name.

Copy & set don't perform type validation

Besides passing values via the constructor, we can also pass values via copy & update or with setters (Pydantic's models are mutable by default). These, however, have surprising behavior.

Copy & update won't perform any type of validation. We can see that in the following example:

1. Create a regular model that coerces input types.
2. Copy `Pizza` with an incompatible input value.
3. Surprisingly, our model is copied “successfully” without any `ValidationError` being raised.

Setting a value is another example where Pydantic doesn’t perform any validations:

1. Once again, Create a regular model that coerces input types.

2. Set an incompatible input value to `toppings_count`.
3. Surprisingly, no `ValidationError` is raised when we set `toppings_count` with a bad value.

Luckily, Pydantic does allow us to fairly easily overcome the aforementioned setter problem:

```python
import pytest
from pydantic import BaseModel, ValidationError


class Pizza(BaseModel):
    toppings_count: int
    size: str

    class Config:  # 1
        validate_assignment = True  # 2


def test_pydantic_allows_setters_conversions():
    pizza = Pizza(toppings_count='4', size='XL')
    assert pizza.toppings_count == 4
    pizza.toppings_count = '5'  # 3
    assert pizza.toppings_count == 5  #  3
    with pytest.raises(ValidationError):  # 4
        pizza.toppings_count = 'no conversion'  # 4
```

validate_assignment_fixes_setter.py hosted with ❤ by **GitHub** **view raw**

1. `Config` inner class defines custom configurations on our models.
2. This is how we tell Pydantic to make our setters perform validations (& type coercion) on inputs.
3. Pydantic now performs type coercion as we would expect (at least as I would expect).
4. Incompatible types raise `ValidationError`s.

I could not find an easy way to do the same for copy & update (aside from rewriting `copy`)

Adding constraints to models

It's easy to start using Pydantic with the known type hints like `str`, `int`, `List[str]`, etc. In many cases though, these types are not enough and we may want to further constrain the types.

Further constraining our models' types is usually advantageous as it tends to reduce the amount of code (and conditionals), fail-fast (usually at our system's boundaries), provides better error handling, and better reflects our domain requirements (There's a very interesting lecture that is related to this called constraints liberate liberties constrain). Pydantic provides several options for adding constraints.

Custom Pydantic types

Pydantic ships with a few useful custom types. Some specific types are:

- URLs — input must match a URL schema. Also has a set of functions to extract the different parts of a URL.
- File paths — input must be a valid existing file.
- UUID — input must represent a valid UUID.
- Secret types — hide the values when printed or when displayed as a JSON.
- Payment card numbers — input must match a payment card number schema. Also provides a way to access the relevant parts of the number (brand, bin, etc).
- Be sure to check the documentation as there are more.

Constraint types

It's possible to define primitive types that have more constraints on their values. These are especially useful to narrow the number of cases our systems need to deal with. Some examples are non-empty strings, non-empty lists, positive ints, a range of numbers, or a string that matches a certain regex.

Consider the following example:

1. `constr` is a type of a constrained `str` — the `str` must have at least 1 character.
2. `conlist` is a type of a constrained `List[int]` — the list must have at least one score.
3. No name validation required since `User` s are guaranteed to have at least one character in the name (assuming non-empty strings are valid names of course).
4. We can immediately use `max` on the `User` 's `scores` , as `scores` guaranteed to be non-empty — I find this the be the most interesting part when using constrained types, we avoid having to deal with many edge cases.

5. Pydantic verifies that `name` is at least one character long.
6. Pydantic verifies that `scores` are not empty.
7. Inputs that don't obey the constraints causes Pydantic to raise a `ValidationError`.

The distinction between custom types & constrained types is that custom types are new types with relevant behavior (e.g. URL has a `host` attribute), while constrained types are just primitive types that can only accept a subset of their inputs' domain (e.g. `PositiveInt` is just an `int` that can only be instantiated from positive `int`s).

Custom validators

When Pydantic's custom types & constraint types are not enough and we need to perform more complex validation logic we can resort to Pydantic's custom validators. These are basically custom validation functions we add to the models.

Enforcing models with types strictness

In order to explain Strict Types let's start with 2 examples:

```python
from pydantic import BaseModel, PositiveInt


class UserResponse(BaseModel):
    user_input: bool  # 1


def test_what_did_the_user_mean():
    assert UserResponse(user_input='yes').user_input is True  # 2


class Summary(BaseModel):
    score: PositiveInt  # 3


def test_summaries_with_different_scores_are_equal():
    assert Summary(score=4.3) == Summary(score=4.4)  # 4

```

ambiguous_type_coercion.py hosted with ❤ by GitHub view raw

1. We expect our users to provide a `bool` response.
2. There **may be** some ambiguity in the user's response. The user wrote "yes" but we interpreted it as `True` (this may be the expected behavior but there's room for error here).
3. Type coercion causes us to lose information causing 2 different summaries to have the same score.

These problems arise from the fact the Pydantic coerces values to the appropriate types. In most cases the type coercion is convenient but in some cases, we may wish to define stricter types that prevent the type coercion.

Here is an example of how this can be done:

```python
import pytest
from pydantic import BaseModel, ValidationError, StrictInt, StrictBool


class UserResponse(BaseModel):
    user_input: StrictBool  # 1


def test_only_bool_responses_are_accepted():
    assert UserResponse(user_input=True).user_input is True  # 2
    assert UserResponse(user_input=False).user_input is False  # 2


@pytest.mark.parametrize(
    'user_input',
    ['yes', 'no', 'true', 'false', 'True', 'False']
)
def test_non_bool_responses_are_not_strict_enough(user_input):
    with pytest.raises(ValidationError):  # 3
        UserResponse(user_input=user_input)  # 3


class Summary(BaseModel):
    score: StrictInt  # 4


@pytest.mark.parametrize(
    'score',
    [3.4, '3', '4.0']
)
def test_only_ints_are_valid_scores(score):
    with pytest.raises(ValidationError):  # 5
        Summary(score=score)  # 5
```

1. We let Pydantic know that `user_input` is a strict boolean type.
2. Only `True` & `False` can be used as inputs for `user_input`.
3. Values that would usually be coerced into `bool` are no longer coerced and result in a `ValidationError` being raised.
4. `score` can now only receive `int`s and no other types.
5. `int` compatible types are no longer coerced and result in a `ValidationError` being raised.

Defining none key-value models

Most of the models we use with Pydantic (and the examples thus far) are just a bunch of key-value pairs. However, not all inputs can be represented by just key-value inputs.

Consider the following withered example:

1. `Names` cannot represent a list of `str` as it must be initialized with the `values` field but the input data doesn't match this expectation — this results in a `TypeError`.
2. Similarly, `Name` cannot be created without using the `value` field name.

In some cases, it's useful to define models that are just specialized representations of primitive types. These specialized types behave just like their primitive counterparts but have a different meaning to our program.

Let's look at how we can achieve this:

```python
from pydantic import BaseModel


class Age(BaseModel):
    __root__: int  # 1


def test_specialized_models_are_not_equal_to_their_primitives():
    age = Age(__root__=42)
    assert age != 42  # 2


def test_age_can_also_be_parsed():
    age = Age.parse_obj(42)  # 3
    assert age != 42


def test_specialized_models_can_ensure_we_represent_the_correct_types():
    age = Age(__root__='43')
    assert age.__root__ == 43  # 4
```

non_key_values.py hosted with ❤ by **GitHub** **view raw**

1. `__root__` is our way to tell Pydantic that our model doesn't represent a regular key-value model.
2. Despite `Age` having a value of 42, it's not equal to a regular primitive 42 `int`.
3. `parse_obj` is just another convenient method to parse inputs.
4. Pydantic maintains type coercion for custom `__root__` models.

These custom `__root__` models can be useful as inputs as we can see in the following example:

```python
from pydantic import BaseModel


class Age(BaseModel):
    __root__: int


def age_in_days(age: Age) -> int:
    return age.__root__ * 365  # 1


def test_age_in_days_only_needs_to_deal_with_ints():
    age = Age.parse_obj('42')  # 2
    assert age_in_days(age) == 42 * 365


# note that mypy raises an error when trying the following
# will raise a mypy error
# age_in_days(age='not an age')  # 3
```

functions_are_dry_with_models.py hosted with ❤ by **GitHub** **view raw**

1. `age_in_days` can only focus on performing days calculation and doesn't require any extra validation or parsing code.
2. `__root__` models perform type coercion just like any other model.
3. If we're naughty and try hard enough we can obviously provide `age_in_days` a non-age value but with mypy we can at least spot some of the typing issues.

Other than what we've already discussed `__root__` models have the following interesting consequences:

1. `foo` expects 2 variables: 1. an `int` representing an `age` and a regular `int`. When invoking `foo` it's easy to accidentally pass the arguments in the wrong order.

2. Luckily, mypy can help spot these errors.

3. Even though `Age` is defined as a custom `__root__` model, when we convert `Person` to JSON, `Age` behaves just like a regular `int`.

So far we've discussed the advantages, there are, however, a few things we should consider:

- Although cool, this can easily be overused and become hard/complicated to use. Part of what makes Python so fun is it's simplicity — be aware and try to avoid overusing this feature.
- Although premature optimization is the root of all evil — using these models in performance-critical sections may become a bottleneck (as we're adding more objects, validations, etc). Be aware of this when aiming for performance (this is also true for "regular" Pydantic models and not just for custom `__root__` models).

Defining these custom `__root__` models can be useful **when used appropriately**.

Settings management with Literal types

There's another useful feature that works with `__root__` but first, let's discuss how Pydantic helps us deal with reading & parsing environment variables.

A lot of code that I've seen around reading & parsing applications settings suffers from 2 main problems: 1. there's a lot of code around reading, parsing & error handling (as environment variables may be missing, misspelled, or with an incompatible value) — these usually come in the form of utility code. 2. when there are multiple errors we will usually start a highly annoying cycle of trying to read the configurations, failing on the first error (program crashes), fixing the error, repeat * N (where N is the number of configuration errors)

These are obviously very annoying, but luckily with Pydantic these problems are very easy to solve using Pydantic's `BaseSettings`. This is more or less all we need to do:

```python
import os
import pytest
from pydantic import BaseSettings, HttpUrl, ValidationError


class Config(BaseSettings):  # 1, 2
    auth_url: HttpUrl
    db_name: str
    max_retries: int

def test_read_and_parse_env_vars():
    os.environ.clear()
    os.environ['auth_url'] = 'http://www.weird.auth.name.com'
    os.environ['db_name'] = 'mydb'
    os.environ['max_retries'] = '7'

    config = Config()  # 3

    assert config.auth_url == 'http://www.weird.auth.name.com'
    assert config.db_name == 'mydb'
    assert config.max_retries == 7


def test_all_errors_appear_together():
    os.environ.clear()
    os.environ['auth_url'] = 'not a url'
    os.environ['max_retries'] = 'not an int'

    with pytest.raises(ValidationError) as e:
        Config()  # 4

    assert 'auth_url' in str(e.value)  # 5
    assert 'max_retries' in str(e.value)  # 5
    assert 'db_name' in str(e.value)  # 5
```

1. Define a Pydantic model with all the required fields and their types.

2. Inherit from Pydantic's BaseSettings to let it know we expect this model to be read & parsed from the environment (or a .env file, etc)

3. Create the model without any input values (values are read from the environment).

4. Since there are errors, trying to read `Config` results in a `ValidationError` being raised.

5. Since Pydantic accumulates errors with `ValidationError` we can see all the errors at once.

`BaseSettings` in itself is a very useful feature but often we need to read different models (different fields & types) where each model is determined by the environment we're running on. How can we achieve this using Pydantic?

This can be achieved by combining Literal Types, Union Types & `__root__` (which we looked at previously). This is the gameplan:

- Define different configuration models (prod/staging/local etc)
- Each configuration model will also include a field `env` (I tend to call these `env` or `profile` but you can choose whatever name you like) with a Literal type representing the name of the corresponding environment.
- Define a configuration union type of all possible configuration models.
- Use `parse_obj_as` to make Pydantic read it according to the actual `ENV` value.

Let's look at some code:

```python
import os
from typing import Union

import pytest
from pydantic import BaseSettings, HttpUrl, ValidationError, parse_obj_as
from typing_extensions import Literal


class LocalContext(BaseSettings):  # 1
    env: Literal["local"]  # 1
    echo_server_url: str


class ProdContext(BaseSettings):  # 1
    env: Literal["prod"]  # 1
    external_server_url: HttpUrl
    external_server_port: int


Context = Union[LocalContext, ProdContext]  # 2


def test_reads_local_context():
    os.environ.clear()
    os.environ["ENV"] = "local"  # 4
    os.environ["ECHO_SERVER_URL"] = "http://localhost"

    context = parse_obj_as(Context, {})  # 3

    assert context.env == "local"  # 4
    assert context.echo_server_url == "http://localhost"
    assert not hasattr(context, "external_server_url")
    assert not hasattr(context, "external_server_port")


def test_reads_prod_context():
    os.environ.clear()
    os.environ["ENV"] = "prod"  # 5
    os.environ["EXTERNAL_SERVER_URL"] = "http://some.echo.serv.er"
    os.environ["EXTERNAL_SERVER_PORT"] = "8080"

    context = parse_obj_as(Context, {})  # 3

    assert context.env == "prod"  # 5
    assert context.external_server_url == "http://some.echo.serv.er"
    assert not hasattr(context, "echo_server_url")


```

```
49  def test_only_given_literals_are_supported():
50      os.environ.clear()
51      os.environ["ENV"] = "staging"
52
53      with pytest.raises(ValidationError):  # 6
54          parse_obj_as(Context, {})  # 6
```

root_with_literals_improved.py hosted with ❤ by GitHub view raw

3. `parse_obj_as` followed by the empty dictionary, is our way to tell Pydantic to read `Context` as settings. Note that since `Context` can either be `LocalContext` or `ProdContext` it must be of type `BaseSettings`. This means we don't need to provide any arguments to `parse_obj_as` when invoking it.

4. `local` matches the literal type `local` and therefore `LocalContext` is read.

5. `prod` matches the literal type `prod` and therefore `ProdContext` is read.

6. If there was a failure reading `Context` we raise a `ValidationError`.

Edit: I initially posted a slightly more complex version of this code but thanks to Nuno André I was able to simplify it.

Note that we obviously still need to programmatically check the `env` variable to know which context we actually read (as it was determined by an environment variable) but:

- We only need to read & parse the context **once** from the environment as opposed to doing this in 2 steps: 1. read only the `env` variable from the environment. 2. read the rest of the environment variables according to the `env` variable.

- We sometimes want to have the environment name available to us for logging, monitoring, etc so having `env` may not be redundant and can actually prove useful.

Using Pydantic to perform functions arguments validation

This feature is very new (still in beta as of the time of writing this) so make sure you read the docs before using this feature in production or rely heavily on it.

So far, we leveraged Pydantic's ability to validate & parse arguments when we used Pydantic models. But what happens when we've got a function that has no Pydantic model as it's arguments but instead only regular arguments? Can we somehow leverage Pydantic to validate these arguments?

This is where validate_arguments comes into play. It's basically a Python decorator we can add to any function with type hints and Pydantic will validate the function arguments (works on methods too).

Let's look at an example:

1. No Pydantic model. Even though `retries` is a `PositiveInt` we won't get any validations.

2. An `AttributeError` is raised since `get_payload` is passed wrong arguments. Also, note the 2nd test case where both arguments are invalid. When this happens we will only get an `AttributeError` on the first argument and not on both.

We can add validations to the function by using `validate_arguments`:

1. In order to coerce input types or fail for invalid inputs, we need to add the `validate_arguments` decorator.

2. Since `validate_arguments` actually performs Pydantic validations on the input, invalid inputs are no longer allowed.

3. Pydantic raises a `ValidationError` on bad inputs.

Not directly related to `validate_arguments` but if we're already using Pydantic we can make the `get_payload` function even better by specifying the types that we actually need like this:

1. `url` moved from `str` to a more specific `HttpUrl` type.

2. Since `HttpUrl` is already a valid URL there's no need to perform checks on it inside the function — once again, having more constrained types helped us remove complexity.

Although new, `validate_arguments` seems like a really nice & useful addition to Pydantic.

Summary

Pydantic is very easy to get started with, but it's also easy to overlook some of it's more useful features. These features are important to know as they can help us improve our overall code quality & have better ways to handle errors — all these with relatively little effort.

Programming Python Python 3 Code Quality Clean Code