Python: Pydantic

Introduction

Python's appeal comes from its easy usability. Unlike more difficult programming languages such as C++ or Rust, Python does not require us to declare our data types before assigning them.

However, when programming data processes and pipelines, the goal is to be as strict as possible with both data types and possible values that variables can accept. This ensures that all processes are functioning as they should be and no faulty, incorrect data seeps through the cracks.

Problem with Assert Statements

Assert statements are a built-in Python functionality used to force conditions to be true and will throw an exception otherwise. For data validation or data processing, it is *not* recommended to use assert statements.
Assert statements can be turned off globally by the Python interpreter via command line arguments or a CPython environment variable.

```
# BAD: Data validation can be skipped entirely
assert isinstance(my_int, int)
assert isinstance(my_string, str)
assert isinstance(df, pl.DataFrame)

# EVEN WORSE: We could insert incorrect or invalid data
assert portfolio_number in valid_portfolio_numbers_list

# THE WORST: Assertions for permissions can easily be bypassed
assert user.is_admin()
```

This is a reason why we rarely ever see assert statements actually being used outside of test frameworks. Alternatively, we could use `isinstance()` to raise errors manually:

```
if not isinstance(my_int, int):
    raise TypeError(f"Expected id to be type int, got {type(id).__name__}")
```

However, type-checking and value-checking every argument and variable would add many lines of tedious, unnecessary code. Instead, Pydantic has become the popular choice for upstream data validation and is a more powerful alternative to native Python dataclasses. Pydantic is a relatively new, popular library used for data validation and increased data type strictness. Like Polars, Pydantic was written in Rust and is exceptionally fast and is used by developers at Apple, Google, OpenAI, JP Morgan, Nvidia, Microsoft, the NSA, NASA, etc.

The Pydantic API documentation can be found here: Pydantic Documentation

Simple Type Checking

Pydantic includes the `@validate_call` decorator which can be placed on functions to turn type-hints into type-checks. If the data type provided does not match, a `ValidationError` will be raised:

```
from pydantic import validate_call

@validate_call
def print_name(my_name: str) -> None:
    print(f"My name is {my_name}")

# This will raise a ValidationError since it type-hints a str
print_name(123)
```

Before raising a `ValidationError`, the `@validate_call` decorator will try to parse the data type if possible. This is particularly useful when, for example, dealing with the different ways of writing dates: `datetime`, `date`, or `str` in ISO-format. This parsing allows all three date data types to be used while maintaining strictness in the code downstream. For example:

```
from datetime import datetime, date
from pydantic import validate_call

@validate_call
def show_date(my_date: date) -> None:
    # Write code for a date object
    ...

# All of these work! Arguments automatically parse
show_date('2024-05-31')
show_date(date(2024, 5, 31))
show_date(datetime(2024, 5, 31))
```

In order to allow arbitrary types (i.e. non-native Python data types) to be type checked, a configuration dictionary, specifically `ConfigDict(arbitrary_types_allowed=True)` must be passed in:

```
import polars as pl
from pydantic import validate_call, ConfigDict

model_config = ConfigDict(arbitrary_types_allowed=True)

@validate_call(config=model_config, validate_return=True)
def wrangle_df(df: pl.DataFrame) -> pl.DataFrame:
    ...
```

For more information on `ConfigDict`, see the section *Configuration* farther below.

This is an easy, standalone way to add type-checking and parsing into Python code. However, the main usage of Pydantic are 'models', particularly BaseModel, which can be thought of as an improvement over native Python dataclasses.

BaseModel

Consider the following code for object instantiation:

```
class Food:
    ...

class User:
    def __init__(self, id: int, name: str = "John Doe", allergies:
list[Food] | None = None):
        if not isinstance(id, int):
            raise TypeError(f"Expected id to be type int, got
{type(id).__name__}")

        if not isinstance(name, str):
            raise TypeError(f"Expected name to be type str, got
{type(name).__name__}")

        if allergies is not None and not isinstance(allergies, list):
            raise TypeError(f"Expected name to be type list, got
{type(allergies).__name__}")

        self.id = id
        self.name = name
        self.allergies = allergies
```

A Pydantic `BaseModel` can simplify and refactor this code by transforming type-hints into type-checks, raising errors if the data type and value requirements are not fulfilled. Arguments will attempt to be parsed as their correct data type before raising a `ValidationError` exception:

```
from pydantic import BaseModel

class Food(BaseModel):  # This class must also inherit BaseModel
    ...

class User(BaseModel):
    id: int
    name: str = "John Doe"
```

```
    allergies: list[Food] | None = None

# WORKS: The string can be parsed as an int
new_user = User(id='123')

# FAILS: Raises a ValidationError
new_user = User(id='abc')
```

Any `ValidationError` exceptions can be recorded using a *try, except* block and the `errors()` method, or `json()` to return it in json format:

```
from pydantic import ValidationError

try:
    new_user = User(id='abc')
except ValidationError as exc:
    exceptions = ex

# Any exceptions are saved as list[dict[str]]
print(exceptions.errors())
```

By inheriting from `BaseModel`, a variety of methods are included and are detailed in the API documentation. Popular ones are showcased below:

```
# Get which fields were set through arguments, not defaults (returns
dict[str])
new_user.model_fields_set

# Get the fields and their values as a dictionary (returns dict[str])
new_user.model_dump()

# Get the fields and their values as a json (returns str)
new_user.model_dump_json()

# Get the json schema of the fields and their values (returns dict[str])
new_user.model_json_schema()
```

Custom Serialization

Pydantic has a protocol to automatically serialize different data types. For example, a `datetime` object is expressed in json format as a string in a certain way:

```
from datetime import datetime
from pydantic import BaseModel
```

```
class MyModel(BaseModel):
    my_datetime: datetime

my_model = MyModel(my_datetime=datetime.now())

# Expressing a datetime as a json is expressed as a string by default
my_model.model_dump_json()      # returns: '{"my_datetime":"2024-02-
09T23:42:54.562087"}'
```

However, we can override serialization behavior using the `@field_serializer()` decorator. In this example, we can choose to serialize the `my_datetime` field as an Epoch Unix timestamp:

```
import time
from datetime import datetime
from pydantic import BaseModel, field_serializer

class MyModel(BaseModel):
    my_datetime: datetime

    @field_serializer('my_datetime', when_used='always')
    def serialize_my_datetime(self, value):
        # Convert the datetime to an Epoch Unix timestamp
        return time.mktime(self.my_datetime.timetuple())

my_model = MyModel(my_datetime=datetime.now())

# All default serialization is now overridden
my_model.model_dump_json()      # returns: '{"my_datetime":1707630174.0}'
my_model.model_dump()           # returns {'my_datetime': 1707630174.0}
```

By default all serialization will be overridden. However, we can adjust the `when_used=` argument to specify when to apply this serialization overriding. Possible arguments include:

- `always` - Override all serializations (default)
- `unless-none` - Override always except for `None`. Useful if the field serializer function cannot handle `None` values.
- `json` - Only override for json serialization
- `json-unless-none` - Only override for json serialization but not `None`. Useful if the field serializer function cannot handle `None` values.

Fields & Constraints

Field Objects

Pydantic includes a variety of tools available to apply constraints to values. Example include the `Field` object or Pydantic object types:

```
from pydantic import BaseModel, Field
from pydantic.types import PositiveInt, conlist

class User(BaseModel):
    # Only allow 'int' to take positive integers
    id: PositiveInt

    # Only allow 'name' to take letters and numbers (specified by RegEx)
    # and 50 characters maximum
    name: str = Field(
        default="John Doe",
        pattern=r"^[a-zA-Z0-9-' ]+$",
        min_length=1,
        max_length=50
    )

    # Do not allow more than 100 allergies
    allergies: conlist(Food, min_length = 1, max_length=100) = []
```

A `Field` object's default value can also take a function, allowing it to be dynamic:

```
from uuid import uuid4
from pydantic import BaseModel, Field

class User(BaseModel):
    id: int = Field(default_factory=lambda: uuid4().hex)
    ...
```

A `Field` object can also be given an 'alias', which allows another alternative name to be linked to the field. This is helpful in cases where the attribute names of two sources differ, such as an API and a database field. If multiple aliases are included, an `AliasChoices` object can be specified to search for each alias left-to-right. However, if an alias is set, they must be used and the original field names cannot be used by default (to change this default setting, `model_config` must be set. See the section **Configuration** for more information):

```
from pydantic import BaseModel, Field, AliasChoices

class User(BaseModel):
    name: str = Field(..., alias='username')
    sex: str = Field(..., alias=AliasChoices(
        'gender',
```

```
        'birth_sex',
        'birth_gender'
    ))
    ...

# WORKS: The 'name' field must now be set by passing in 'username'; same
with 'sex'
new_user = User(username="John Doe", birth_sex='Male')

# FAILS: Once aliases are set, they must be used
new_user = User(name="John Doe", sex='Male')

# We can specify whether to dump with the aliases or original field names
new_user.model_dump(by_alias=True)      # returns: {'username': 'John Doe',
'birth_sex': 'Male'}
new_user.model_dump(by_alias=False)     # returns: {'name': 'John Doe',
'sex': 'Male'}
```

A `Field` object also includes a variety of arguments to provide constraints for possible values:

```
from decimal import Decimal
from pydantic import BaseModel, Field

class User(BaseModel):
    # Length between 3 and 10 characters; only letters and numbers
    username: str = Field(..., min_length=3, max_length=10,
pattern=r'^\w+$')

    # Between 1 and 120
    age: int = Field(..., gt=0, le=120)

    # Maximum of 10 digits and 2 decimal places
    balance: Decimal = Field(..., max_digits=10, decimal_places=2)

    # Minimum of 1 value
    allergies: List[Food] = Field(..., min_items=1)
```

Field Decorators

The `@computed_field` decorator can be used to create a `Field` object out of a property attribute. This allows for fields to be set based on more complex calculations:

```
from datetime import date
from pydantic import BaseModel, computed_field
```

```python
class Person(BaseModel):
    name: str
    birth_year: int

    @computed_field
    @property
    def age(self) -> int:
        current_year: int = date.today().year
        return current_year - self.birth_year

# The attribute 'age' is now available
new_person = Person(name="John Doe", birth_year=2000)
print(new_person.age)
```

Fields can undergo more advanced validation and transformation using the `@field_validator()` decorator. For example, if we wanted to force a `name` field to include a space to include both a first and last name, as well as capitalize then entire string:

```python
from pydantic import BaseModel, field_validator

class User(BaseModel):
    ...
    name: str

    @field_validator('name')
    @classmethod
    def name_must_contain_space(cls, value: str) -> str:
        """
        Validates that the 'name' field contains a space and forces upper-
case.
        """
        if ' ' not in value:
            raise ValueError("The 'name' field must contain a space.")
        else:
            return value.upper()

# FAILS: A space does not exist
new_user = User(..., name = "John")

# WORKS: A space exists in the field provided; 'name' is now "JOHN DOE"
new_user = User(..., name = "John Doe")
```

Rather than performing validation on specified fields, more validation and transformation can be performed on the entire model using the `@model_validator()` decorator.

The example below shows how we can include more customized code to control the data that is passed in:

```
from typing import Any
from pydantic import BaseModel, model_validator

class User(BaseModel):
    ...
    name: str

    # This method will check data BEFORE instantiation
    @model_validator(mode='before')
    @classmethod
    def check_sensitive_info_omitted(cls, data: Any) -> Any:
        """
        Ensures that sensitive info, such as 'password' or 'salary', are
omitted.
        """
        if isinstance(data, dict):
            if 'password' in data:
                raise ValueError("Argument 'password' is sensitive and
should not be included.")
            if 'salary' in data:
                raise ValueError("Argument 'salary' is sensitive and should
not be included.")
                return data

        # This method will check data AFTER instantiation
        @model_validator(mode='after')
        def name_must_contain_space(self) -> 'User':
            """
            Validates that the 'name' field contains a space.
            """
            if ' ' not in self.name:
                raise ValueError("The 'name' field must contain a space.")
            else:
                return self.name.upper()

# FAILS: Don't include a 'password' field!
new_user = User(..., password="password123")

# WORKS: No sensitive info was provided
new_user = User(...)
```

Strict Mode

An alternative to instantiation is to use the `BaseModel` method `model_validate()`, in which one could also pass in a dictionary or a string in json format:

```
# As seen before:
new_user = User(id=123, name="John Doe")

# A dictionary alternative:
new_user = User.model_validation({'id': 123, 'name': 'John Doe'})

# A json alternative
new_user = User.model_validation("""{"id": 123, "name": "John Doe"}""")
```

By default, Pydantic tries to parse values to the declared data type. However, the `model_validate()` method can be specified to be 'strict' and not allow any parsing to take place:

```
# WORKS: 'id' can be parsed as an int
new_user = User(id='123', name="John Doe")

# FAILS: 'strict=True' prevents parsing
new_user = User.model_validation({'id': '123', 'name': 'John Doe'},
strict=True)
```

Configuration

While `Field` object provide arguments to set constraints or behaviors on a per-field basis, they can also be set across the entire Pydantic model by provided a `model_config = ConfigDict(...)` field; the field must use this exact name in order to be recognized an omitted as a class attribute:

```
from pydantic import BaseModel, ConfigDict

class User(BaseModel):
    model_config = ConfigDict(str_to_upper=True, str_max_length=20)
    name: str

# WORKS: The 'name' field is now upper-case
new_user = User(name="John Doe")        # becomes "JOHN DOE"

# FAILS: The 'name' field is a string that is too long
new_user = User(name='a'*25)
```

`ConfigDict()` also includes an `extra` argument that is used to control how extra attributes are handled during instantiation:

- `allow` - Allow and include any extra attributes
- `forbid` - Forbid any extra attributes by raising a `ValidationError`
- `ignore` - Allow but ignore any extra attributes

For example:

```
from pydantic import BaseModel, ConfigDict

class User(BaseModel):
    model_config = ConfigDict(extra='forbid')
    name: str

# FAILS: Extra attributes is forbidden
new_user = (name="John Doe", age=20)
```

There are a variety of additional arguments, such as `populate_by_name`, which toggles whether to allow fields to be populated by their names, even after specifying aliases. For a full list of potential arguments, please refer to the Pydantic API Documentation: [Pydantic API Documentation: Configuration]

Model configuration can also be set up to have global behavior by creating a `BaseModel` and inheriting from it:

```
from pydantic import BaseModel, ConfigDict

class Config(BaseModel):
    model_config = ConfigDict(extra='allow')

class User(Config):
    name: str
    ...
```

Pydantic Settings

Pydantic Settings provides optional features for loading a settings from environment variables. It is an optional dependency and can be installed with PIP:

```
pip install pydantic-settings
```

The `BaseSettings` class is used to get local environment variables. One method is to specify environment variables in a file called `.env` and attributes of the same name in the Python class:

```
AUTH_KEY=12345ABCDEF
MY_API_KEY=67890WXYZ
```

```
from pydantic import Field
from pydantic_settings import BaseSettings
```

```
class Settings(BaseSettings):
    auth_key: str
    api_key: str = Field(alias='my_api_key')

# This reads the .env file and matches field~environment names
my_settings = Settings()
my_settings.model_dump()        # returns: {'auth_key': '12345ABCDEF',
'api_key': '67890WXYZ'}
```

Another method is to specify environment variables directly in the Python code using `os.environ`:

```
import os
from pydantic import Field
from pydantic_settings import BaseSettings

os.environ['AUTH_KEY'] = '12345ABCDEF'
os.environ['MY_API_KEY'] = '67890WXYZ'

class Settings(BaseSettings):
    auth_key: str
    api_key: str = Field(alias='my_api_key')
```

Environment variables will likely be different between 'development' vs 'production' environments. The `model_config` attribute that takes a `SettingsConfigDict` object can be used to control different environments:

```
DEV_SERVER=MyServer-DEV
PRD_SERVER=MyServer-PRD
```

```
from pydantic_settings import BaseSettings, SettingsConfigDict

class Settings(BaseSettings):
    model_config = SettingsConfigDict(
        env_file='servers.env',
        env_file_encoding='utf-8',
        env_prefix='DEV_',
        case_sensitive=False,
        validate_default=True,
        extra='ignore'
    )
    server: str

# 'SettingsConfigDict' will read the SERVER variable with the prefix 'DEV_'
```

```
my_settings = Settings()
my_settings.model_dump()        # returns: {'server': 'MyServer-DEV'}
```

As shown in the example, `SettingsConfigDict` has a variety of arguments that can control how the file is read and how variables are validated.