Instructions

- Open a terminal and type:

```
git clone http://github.com/DrPaulSharp/pydantic_workshop.git
cd pydantic_workshop
python -m venv pydantic_workshop
```

- Activate the virtual environment

```
pip install -r requirements.txt
```

- Open your favourite editor and get ready to code!

Science and
Technology
Facilities Council

ISIS Neutron and
Muon Source

Science and
Technology
Facilities Council

ISIS Neutron and
Muon Source

Pydantic: A Package for Picky Python Programmers

Paul Sharp
ISIS Research Software Engineering Team
January 2024

@DrPaulSharp

Pydantic

Pydantic is the most widely used data validation library for python.

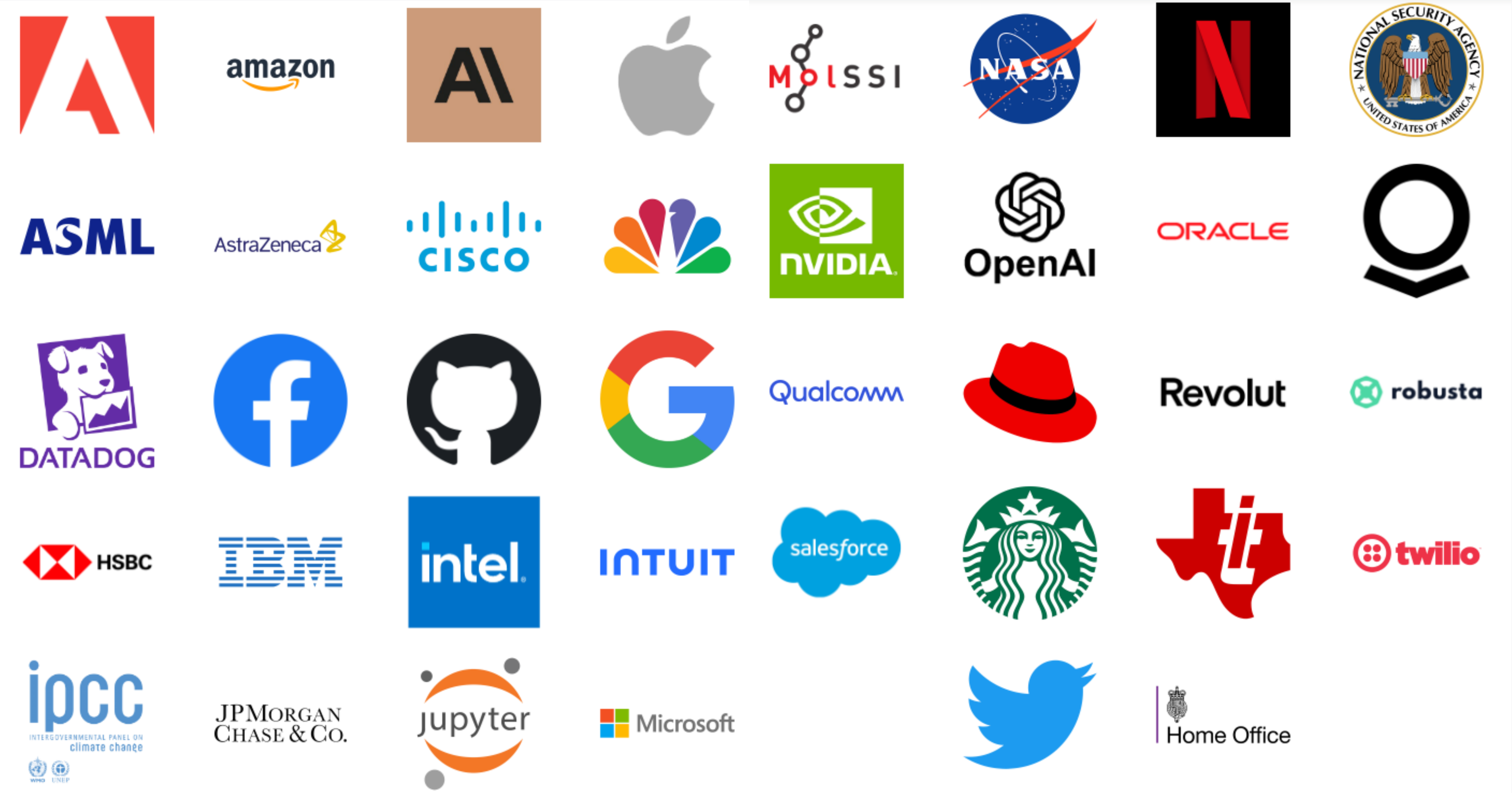
Pydantic is powered by type hints, so data can be defined in pure canonical python 3.7+ and validated with Pydantic.

Pydantic models also share many similarities with Python’s dataclasses.

The docs are available here: https://docs.pydantic.dev/latest/

Pydantic

Pydantic is downloaded 136M times a month, and used by some of the largest and most recognisable organisations in the world, including:



Version 2 or Not Version 2?

Pydantic version 2 was released in June 2023.
Always use version 2+ !

Version 2 introduced a considerable number of changes to the syntax and methods, meaning some examples will be out of date.

Version 2 is also 4x-50x faster than version 1.9.1.

Use the migration guide to ensure everything is up to date.

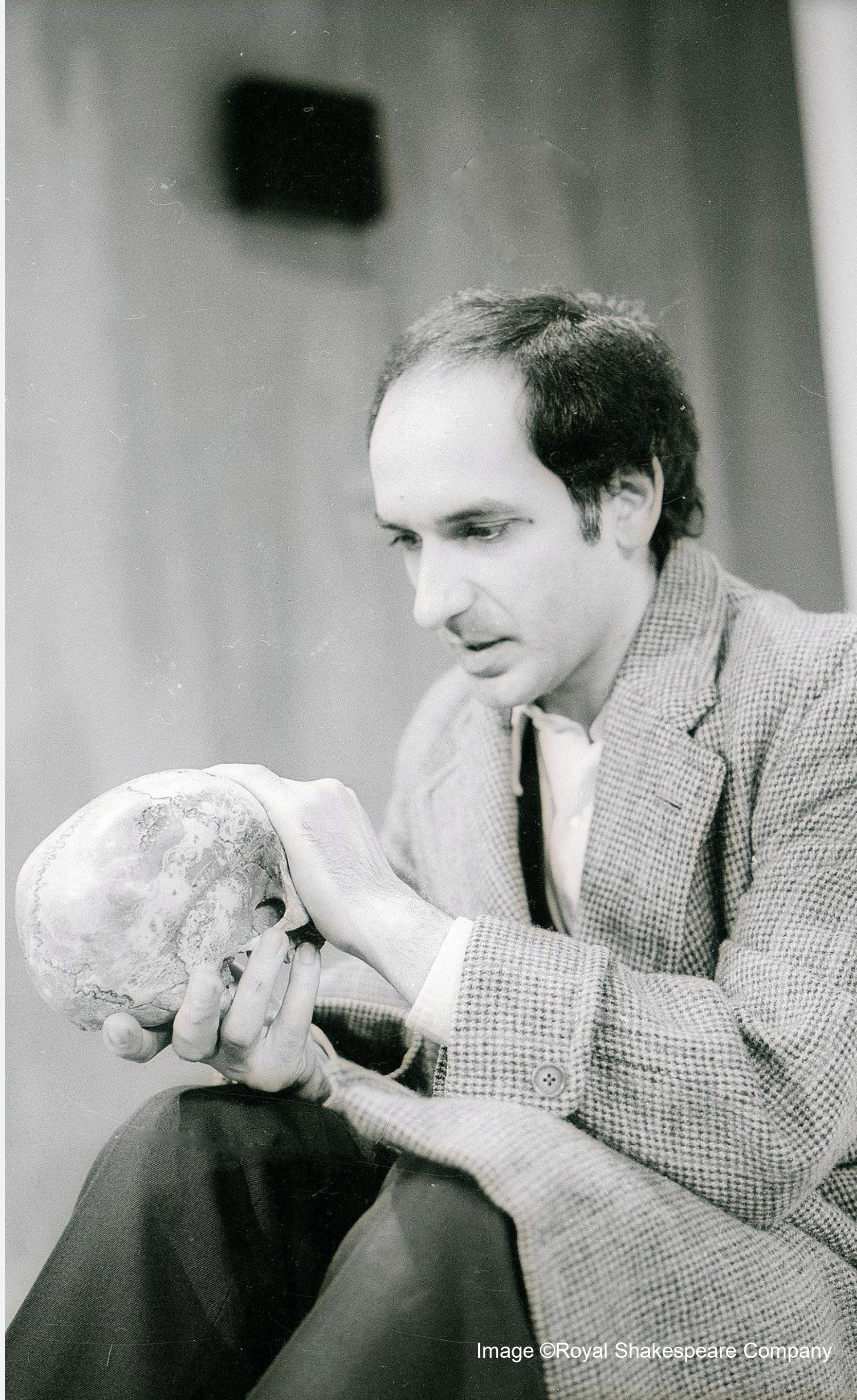


Image ©Royal Shakespeare Company

Type Hinting

Python is dynamically typed – so types of variables are determined only at runtime, and the type of a variable is allowed to change over its lifetime.

PEP 484 introduces type hinting – which supports including type hints in class, function, variable definitions.

These types are not be enforced – they serve as an aid to developers, and can be picked up by an IDE, e.g., pycharm.

```python
def greeting(name: str) -> str:
    return 'Hello ' + name
```

Type Hinting

While these annotations are available at runtime through the usual `__annotations__` attribute, **no type checking happens at runtime**.

Instead, the proposal assumes the existence of a separate off-line type checker which users can run over their source code voluntarily. Essentially, such a type checker acts as a very powerful linter.

Type hints are inspired by the static type checker mypy.

PEP 484 also introduces the typing module, which adds support for type hints.

Classes

```python
class InventoryItem:
    '''Class for keeping track of an item in inventory.'''

    def __init__(self, name: str, unit_price: float, quantity_on_hand: int = 0) -> None:
        self.name = name
        self.unit_price = unit_price
        self.quantity_on_hand = quantity_on_hand

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

Dataclasses

Dataclasses simplify the definition of classes.
They were introduced in PEP 557.

```python
from dataclasses import dataclass

@dataclass
class InventoryItem:
    '''Class for keeping track of an item in inventory.'''
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

Dataclasses

Type hints are required for each field – though can be `typing.Any`.
The type hints are still hints – they are not enforced.

By using a dataclass, we automatically generate the `__init__` method, alongside the following:

```
__repr__
__eq__
__ne__
__lt__
__le__
__gt__
__ge__
```

Pydantic

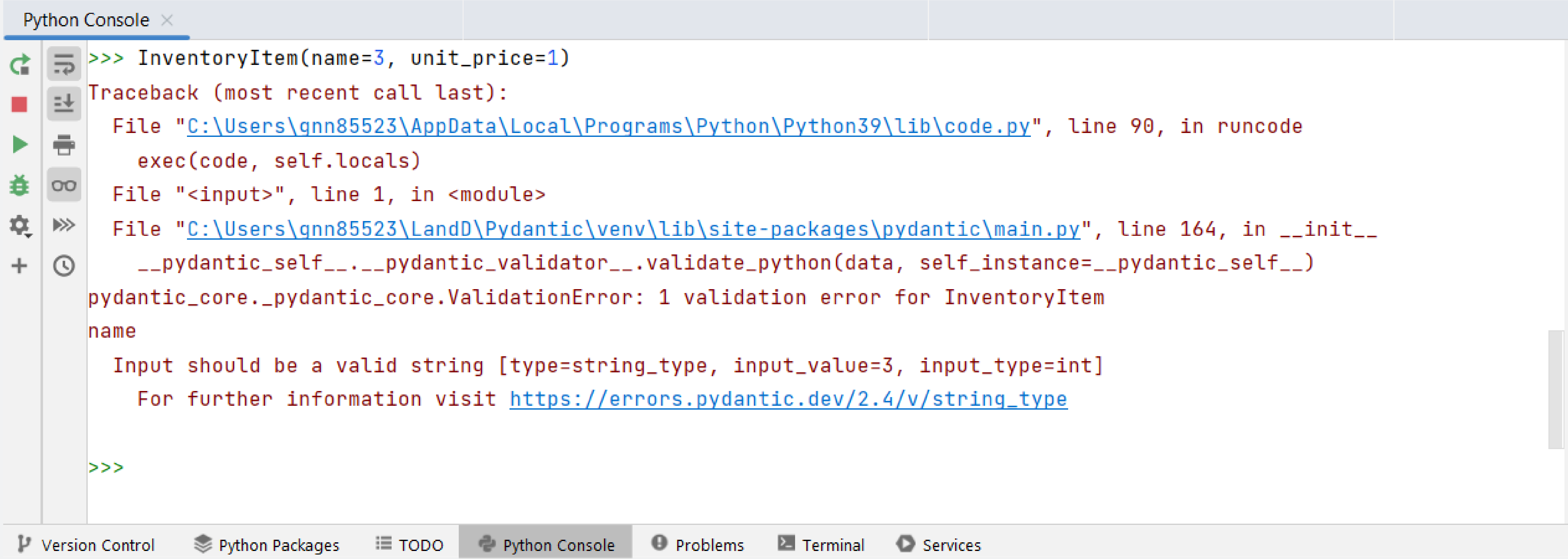
With pydantic, we can enforce our type hints. When a class is defined that inherits from the pydantic `BaseModel`, an error is raised if one of the fields is of the incorrect type.

```python
from pydantic import BaseModel

class InventoryItem(BaseModel):
    '''Class for keeping track of an item in inventory.'''
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

Pydantic



```
>>> InventoryItem(name=3, unit_price=1)
Traceback (most recent call last):
  File "C:\Users\gnn85523\AppData\Local\Programs\Python\Python39\lib\code.py", line 90, in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
  File "C:\Users\gnn85523\LandD\Pydantic\venv\lib\site-packages\pydantic\main.py", line 164, in __init__
    __pydantic_self__.__pydantic_validator__.validate_python(data, self_instance=__pydantic_self__)
pydantic_core._pydantic_core.ValidationError: 1 validation error for InventoryItem
name
  Input should be a valid string [type=string_type, input_value=3, input_type=int]
    For further information visit https://errors.pydantic.dev/2.4/v/string_type

>>>
```

Science and
Technology
Facilities Council

ISIS Neutron and
Muon Source

Napoleon

“The English are a nation of shopkeepers.”

Thought to have been attributed to Napoleon rather than originating from him.

It seems like software developers are a group of shopkeepers as well . . .

Science and Technology Facilities Council

ISIS Neutron and Muon Source

Pydantic Shopkeepers

Let’s run a coffee shop!

Construct a Pydantic model (inherit from `BaseModel`) for a coffee order.

Use the fields: country (str), method (str), size (str), milk (bool), cream (bool), sugars (int) with appropriate defaults.

	TRADITIONAL OR POUR OVER	AEROPRESS	CHEMEX (SERVES TWO)
BEANS A	£2.85	£2.85	£6.50
BEANS B (SEE BOARD)	£3.50	£3.50	£7.25

COE COFFEE & ANY METHOD £10.00

AUGUSTINER HELLES 50cl (5.2%) Augustiner's most popular light lager beer	£4.25
AUGUSTINERBRAU EDELSTOFF 50cl (5.6%) A brighter, sweeter, more sparkly lager	£4.25
KROMBACHER PILS 50cl (4.8%) Finely bitter taste with a full flavoured aroma	£3.50
JEVER 50cl (4.9%) Bitter & hoppy pilsner	£3.50
SPECIALLY SELECTED WEIZEN BEER Naturally cloudy & fruity	£3.75
BROOKLYN LAGER 35.5cl (5.1%) Caramel malts with a floral hop aroma	£2.95
FLYING DOG SNAKE DOG IPA 35.5cl (7.1%) Citrus hop aroma with caramel malt notes	£4.00
FLYING DOG DOGGIE STYLE 35.5cl (5.5%) Grassy, citrus aromas with a sweet body	£3.50
ODELL CUT THROAT PORTER 35.5cl (4.8%) Smooth & robust	£4.00
ODELL 5 BARREL ALE 35.5cl (5.2%) Fresh lively flavour & aroma	£4.00
ODELL IPA 35.5cl (7%) Distinctive & incredible character	£4.50
CIDER	
GWYNT AUTUMN MAGIC MEDIUM 50cl (4.%) Bursts with an autumnal fruity aroma	£3.75
GWYNT BLACK DRAGON MED/DRY 50cl (6.5%) Award winning	£3.75
GWYNT ORCHARD GOLD MEDIUM 50cl (4.9%) Smooth flavour with a fresh crisp sharpness	£3.50
DE TROCH PEACH 25cl (3.5%)	£2.75
DE TROCH STRAWBERRY 25cl (3.5%) Belgium fruit beers	£2.75
HOOK NORTON OLD HOOKY 50cl (4.6%) Beautifully balanced fruity beer	£3.50
RUDGATE BATTLE AXE 50cl (4.8%) Smooth & robust bitter	£3.25
RUDGATE RUBY MILD 50cl (4.4%) Soft rich complex	£3.25
RUDGATE VIKING 50cl (3.8%) Sweet with honey & pineapple notes	£3.25
NAYLORS PINNACLE BITTER 50cl (3.9%) Bitterness with delicate aromas	£2.95
PERONI RED 33cl (4.7%)	£2.75
SUPER BOCK 33cl (5.6%) Refreshing smooth lagers for all occasions	£2.75

MANY SEASONAL SPECIALS
ASK FOR DETAILS

Image ©Restaurant Guru

Science and Technology Facilities Council

ISIS Neutron and Muon Source

Pydantic Shopkeepers

Try some inputs, for example:

```
>> Coffee(country=“Brazil”, milk=False, sugars=0)
>> Coffee(cream=“some”, sugars=“none”)
>> Coffee(method=“pour over”, sugars=“1”)
>> Coffee(size=“small”, milk=“yes”, sugars=1.0)
>> Coffee(country=“Wakanda”, milk=+1, sugars=-1)
```

What do you notice?

Now try with:

```
class Coffee(BaseModel, strict=True):
```

THE ATTIC coffee

STEP 1. CHOOSE BEANS

STEP 2. CHOOSE METHOD

STEP 3. YOUR COFFEE IS PREPARED

STEP 4. ENJOY!

	TRADITIONAL OR POUR OVER	AEROPRESS	CHEMEX (SERVES TWO)
BEANS A	£2.85	£2.85	£6.50
BEANS B (SEE BOARD)	£3.50	£3.50	£7.25

COE COFFEE & ANY METHOD £10.00

BOTTLE BAR menu

AUGUSTINER HELLES 50cl (5.2%) Augustiner's most popular light lager beer	£4.25
AUGUSTINERBRAU EDELSTOFF 50cl (5.6%) A brighter, sweeter, more sparkly lager	£4.25
KROMBACHER PILS 50cl (4.8%) Finely bitter taste with a full flavoured aroma	£3.50
JEVER 50cl (4.9%) Bitter & hoppy pilsner	£3.50
SPECIALLY SELECTED WEIZEN BEER Naturally cloudy & fruity	£3.75
BROOKLYN LAGER 35.5cl (5.1%) Caramel malts with a floral hop aroma	£2.95
FLYING DOG SNAKE DOG IPA 35.5cl (7.1%) Citrus hop aroma with caramel malt notes	£4.00
FLYING DOG DOGGIE STYLE 35.5cl (5.5%) Grassy, citrus aromas with a sweet body	£3.50
ODELL CUT THROAT PORTER 35.5cl (4.8%) Smooth & robust	£4.00
ODELL 5 BARREL ALE 35.5cl (5.2%) Fresh lively flavour & aroma	£4.00
ODELL IPA 35.5cl (7%) Distinctive & incredible character	£4.50
CIDER	
GWYNT AUTUMN MAGIC MEDIUM 50cl (4.%) Bursts with an autumnal fruity aroma	£3.75
GWYNT BLACK DRAGON MED/DRY 50cl (6.5%) Award winning	£3.75
GWYNT ORCHARD GOLD MEDIUM 50cl (4.9%) Smooth flavour with a fresh crisp sharpness	£3.50
DE TROCH PEACH25cl (3.5%)	£2.75
DE TROCH STRAWBERRY25cl (3.5%) Belgium fruit beers	£2.75
HOOK NORTON OLD HOOKY50cl (4.6%) Beautifully balanced fruity beer	£3.50
RUDGATE BATTLE AXE 50cl (4.8%) Smooth & robust bitter	£3.25
RUDGATE RUBY MILD 50cl (4.4%) Soft rich complex	£3.25
RUDGATE VIKING 50cl (3.8%) Sweet with honey & pineapple notes	£3.25
NAYLORS PINNACLE BITTER 50cl (3.9%) Bitterness with delicate aromas	£2.95
PERONI RED 33cl (4.7%)	£2.75
SUPER BOCK 33cl (5.6%) Refreshing smooth lagers for all occasions	£2.75

MANY SEASONAL SPECIALS
ASK FOR DETAILS

Image ©Restaurant Guru

Science and Technology Facilities Council

ISIS Neutron and Muon Source

Basic Models

- Pydantic gives a list of errors – specifically `pydantic.ValidationError`

- Some unusual inputs are accepted – Pydantic “coerces” inputs to the right type, e.g., float converted to int and vice versa, “yes” (or “y”) is a synonym of True etc. See pydantic’s guide to standard library types. (Pydantic can be run in `strict` mode to disable this).

- Some valid but not sensible inputs are accepted – the fictional country of Wakanda, a negative amount of sugar.

We need to further constrain the fields of our model.

Multiple Choice

We can use `typing.Literal` to specify an allowed set of options for a field.

```python
from pydantic import BaseModel
from typing import Literal

class Coffee(BaseModel):
    """Processes a coffee order at the Sharp Coffee Residence."""
    country: Literal["Tanzania", "Ethiopia", "Angola"] = "Angola"
    ...
```

Multiple Choice

Alternatively, we can define an Enum:

```
from pydantic import BaseModel
try:
    from enum import StrEnum
except ImportError:
    from strenum import StrEnum

class Countries(StrEnum):
    Tanzania = 'Tanzania'
    Ethiopia = 'Ethiopia'
    Angola = 'Angola'

class Coffee(BaseModel):
    """Processes a coffee order at the Sharp Coffee Residence."""
    country: Countries = Countries.Angola
```

Literal or Enum?

Both options do the job we want.

The advantage of `typing.Literal` over Enums is performance - ~3x faster.

The advantage of Enums over `typing.Literal` is reusability between different Pydantic models and elsewhere in the code.

Use whichever is best for your purpose.

Field Function

The field function allows for further customisation and validation.

The basic syntax is:

```python
from pydantic import BaseModel, Field
import datetime

current_year = datetime.date.today().year

class User(BaseModel):
    name: str = Field(default='John Doe', min_length=1)
    age: int = Field(..., ge=0)
    birth_year: int = Field(..., le=current_year)
```

Field Function – Keyword Arguments

The field function allows for further validation.

For numeric fields:

- `gt` - greater than
- `lt` - less than
- `ge` - greater than or equal to
- `le` - less than or equal to
- `multiple_of` - a multiple of the given number
- `allow_inf_nan` - allow `'inf'`, `'-inf'`, `'nan'` values

Field Function – Keyword Arguments

The field function allows for further validation.

For string fields:

- `min_length` - Minimum length of the string
- `max_length` - Maximum length of the string
- `pattern` - A regular expression that the string must match

The `min_length` and `max_length` arguments are also useful for fields that require list input.

Field Function – Keyword Arguments

The field function allows for further validation.

For decimal fields:

- `max_digits` - Maximum number of digits within the decimal
- `decimal_places` - Maximum number of decimal places

Field Validators

Field validators are user-written functions that enable further validation for individual fields.

Field validators are class methods rather than instance methods.

A field validator should either return the field or raise a `ValueError` or `AssertionError` (which can be with an `assert` statement).

The same field validator can be applied on multiple fields, but the validator receives only one field of the model as input.

Field Validators

We will focus on “after” validators, which use the following syntax:

```python
from pydantic import BaseModel, Field, field_validator

class User(BaseModel):
    name: str = Field(default='John Doe', min_length=1)
    age: int = Field(..., ge=0)
    birth_year: int = Field(..., le=current_year)

    @field_validator('name')
    @classmethod
    def name_must_contain_space(cls, field: str) -> str:
        if ' ' not in field:
            raise ValueError('Name must contain a space')
        return field
```

Annotated Validators

Annotated validators have similar functionality to field validators but allow for greater reusability between Pydantic models.

They use `typing.Annotated` to apply a validator to a type.

In particular, they can be used to define custom types which Pydantic can validate against.

Further details about custom types in Python are set out in PEP 593.

Annotated Validators

```
from pydantic import BaseModel, Field
from pydantic.functional_validators import AfterValidator
from typing import Annotated

def name_must_contain_space(field: str) -> str:
    if ' ' not in field:
        raise ValueError('Name must contain a space')
    return field

str_with_space = Annotated[str, AfterValidator(name_must_contain_space)]

class User(BaseModel):
    name: str_with_space = Field(default='John Doe', min_length=1)
    age: int = Field(..., ge=0)
    birth_year: int = Field(..., le=current_year)
```

Model Validators

Model validators are like field validators, but apply to the whole model, and are executed when all of the fields have been validated and the model constructed.

They are usually used to resolve the dependence of one field on another.

We will focus on “after” validators, which use the following syntax:

Model Validators

```python
from pydantic import BaseModel, Field, model_validator
import datetime

current_year = datetime.date.today().year

class User(BaseModel):
    name: str = Field(default='John Doe', min_length=1)
    age: int = Field(..., ge=0)
    birth_year: int = Field(..., le=current_year)

    @model_validator(mode='after')
    def check_birth_year_matches_age(self) -> 'User':
        if (self.birth_year != (current_year - self.age) and
            self.birth_year != (current_year - self.age - 1)
        ):
            raise ValueError('Age does not match birth year')
        return self
```

Coffee Validation

Use the Field function to ensure we cannot have a negative amount of sugar.

Use the Literal/Enums to define a set of countries, the coffee methods (traditional, aeropress, pour over and chemex) and sizes.

If the mood takes you . . . come up with some custom names for coffee sizes.

	TRADITIONAL OR POUR OVER	AEROPRESS	CHEMEX (SERVES TWO)
BEANS A	£2.85	£2.85	£6.50
BEANS B	£3.50	£3.50	£7.25
(SEE BOARD)			

COE COFFEE & ANY METHOD £10.00

BOTTLE BAR menu

AUGUSTINER HELLES 50cl (5.2%) £4.25
Augustiner's most popular light lager beer
AUGUSTINERBRAU EDELSTOFF 50cl (5.6%) £4.25
A brighter, sweeter, more sparkly lager

KROMBACHER PILS 50cl (4.8%) £3.50
Finely bitter taste with a full flavoured aroma
JEVER 50cl (4.9%) £3.50
Bitter & hoppy pilsner
SPECIALLY SELECTED WEIZEN BEER £3.75
Naturally cloudy & fruity

BROOKLYN LAGER 35.5cl (5.1%) £2.95
Caramel malts with a floral hop aroma

FLYING DOG SNAKE DOG IPA 35.5cl (7.1%) £4.00
Citrus hop aroma with caramel malt notes
FLYING DOG DOGGIE STYLE 35.5cl (5.5%) £3.50
Grassy, citrus aromas with a sweet body

ODELL CUT THROAT PORTER 35.5cl (4.8%) £4.00
Smooth & robust
ODELL 5 BARREL ALE 35.5cl (5.2%) £4.00
Fresh lively flavour & aroma
ODELL IPA 35.5cl (7%) £4.50
Distinctive & incredible character

CIDER
GWYNT AUTUMN MAGIC MEDIUM 50cl (4.%) £3.75
Bursts with an autumnal fruity aroma
GWYNT BLACK DRAGON MED/DRY 50cl (6.5%) £3.75
Award winning
GWYNT ORCHARD GOLD MEDIUM 50cl (4.9%) £3.50
Smooth flavour with a fresh crisp sharpness

DE TROCH PEACH 25cl (3.5%) £2.75
DE TROCH STRAWBERRY 25cl (3.5%) £2.75
Belgium fruit beers

HOOK NORTON OLD HOOKY 50cl (4.6%) £3.50
Beautifully balanced fruity beer

RUDGATE BATTLE AXE 50cl (4.8%) £3.25
Smooth & robust bitter
RUDGATE RUBY MILD 50cl (4.4%) £3.25
Soft rich complex
RUDGATE VIKING 50cl (3.8%) £3.25
Sweet with honey & pineapple notes
NAYLORS PINNACLE BITTER 50cl (3.9%) £2.95
Bitterness with delicate aromas

PERONI RED 33cl (4.7%) £2.75
SUPER BOCK 33cl (5.6%) £2.75
Refreshing smooth lagers for all occasions

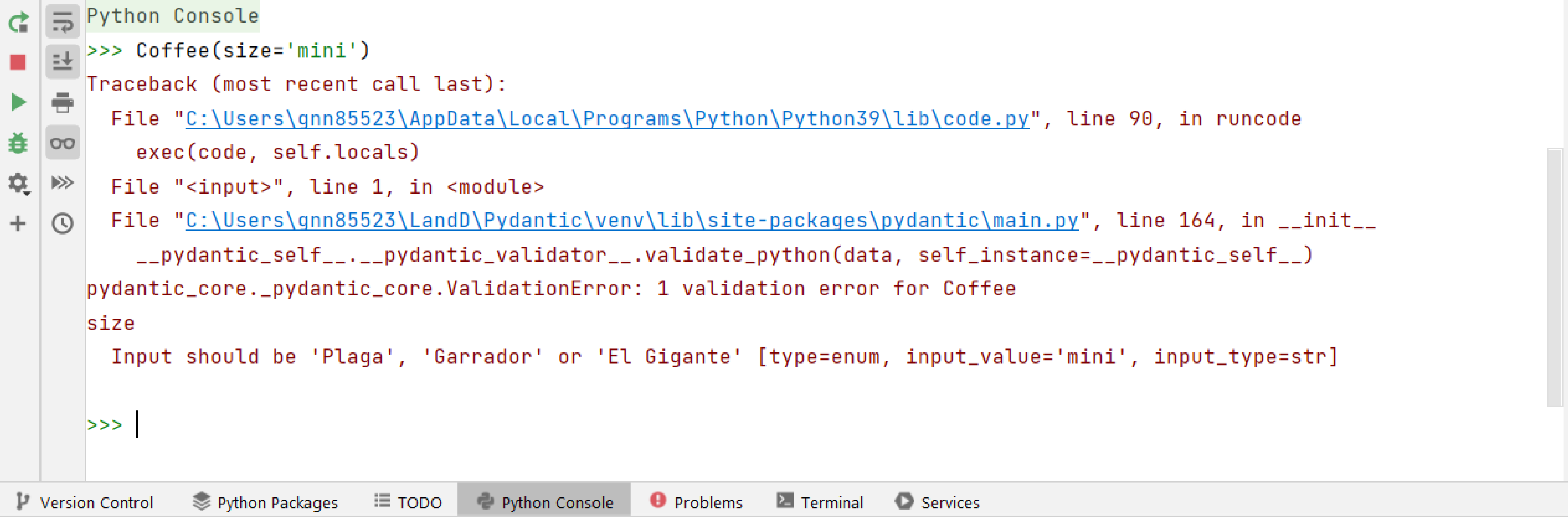
MANY SEASONAL SPECIALS
ASK FOR DETAILS

Image ©Restaurant Guru

Science and Technology Facilities Council

ISIS Neutron and Muon Source

Coffee Size



```
Python Console
>>> Coffee(size='mini')
Traceback (most recent call last):
  File "C:\Users\gnn85523\AppData\Local\Programs\Python\Python39\lib\code.py", line 90, in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
  File "C:\Users\gnn85523\LandD\Pydantic\venv\lib\site-packages\pydantic\main.py", line 164, in __init__
    __pydantic_self__.__pydantic_validator__.validate_python(data, self_instance=__pydantic_self__)
pydantic_core._pydantic_core.ValidationError: 1 validation error for Coffee
size
  Input should be 'Plaga', 'Garrador' or 'El Gigante' [type=enum, input_value='mini', input_type=str]

>>> 
```

Science and Technology Facilities Council

ISIS Neutron and Muon Source

Coffee Validation

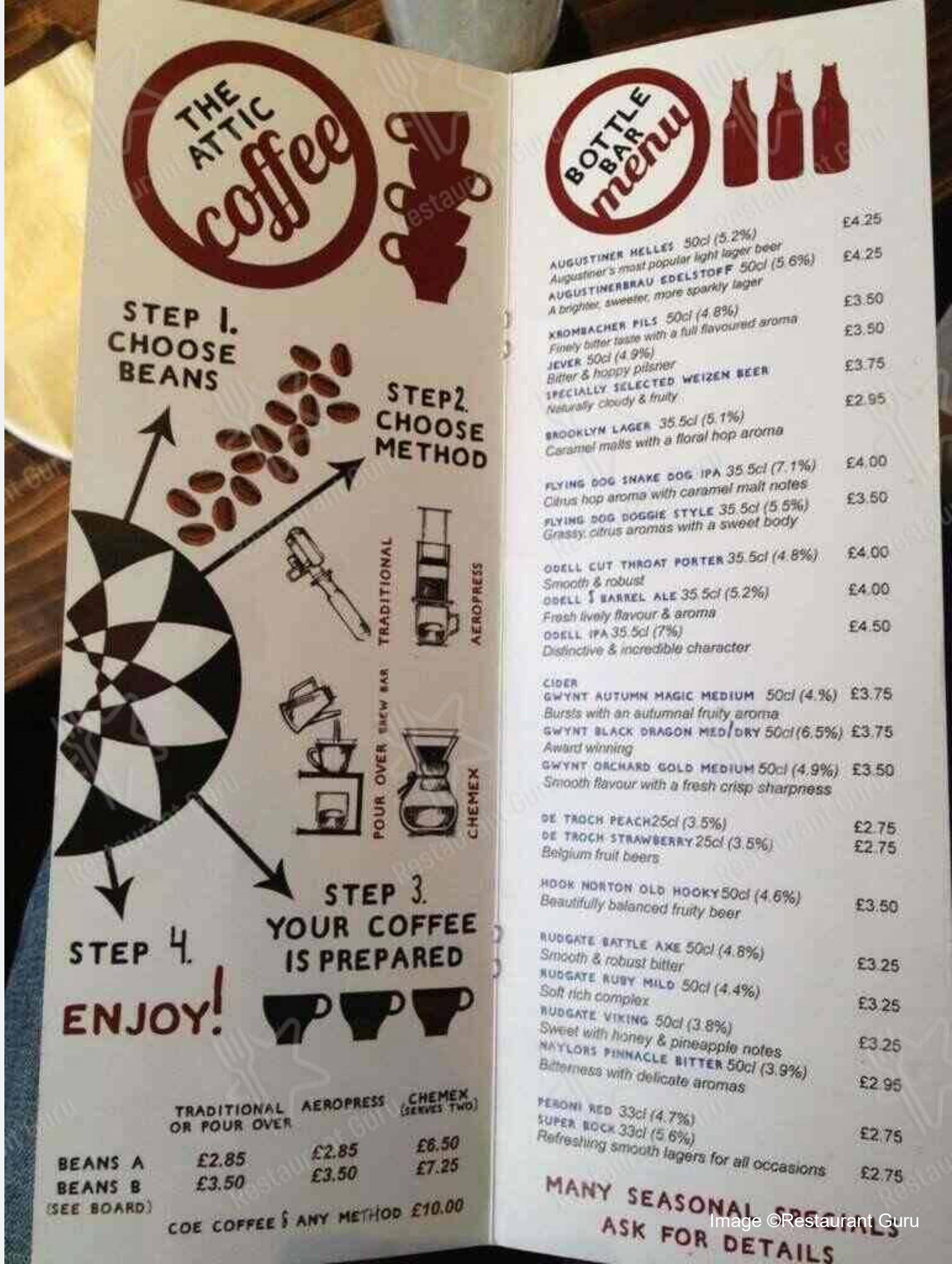
Use validators to make sure that:

- we can’t have milk and cream together,
- we can only have an odd number of sugars (or zero),
- the chemex method is only available for the largest size coffee.

Extra: Create a UUID for an order number field. What’s the default? How do we prevent it from being changed . . .

Science and Technology Facilities Council

ISIS Neutron and Muon Source



THE ATTIC coffee

STEP 1. CHOOSE BEANS

STEP2. CHOOSE METHOD

TRADITIONAL

AEROPRESS

POUR OVER BREW BAR

CHEMEX

STEP 3. YOUR COFFEE IS PREPARED

STEP 4. ENJOY!

	TRADITIONAL OR POUR OVER	AEROPRESS	CHEMEX (SERVES TWO)
BEANS A	£2.85	£2.85	£6.50
BEANS B (SEE BOARD)	£3.50	£3.50	£7.25

COE COFFEE & ANY METHOD £10.00

BOTTLE BAR menu

AUGUSTINER HELLES 50cl (5.2%) Augustiner's most popular light lager beer	£4.25
AUGUSTINERBRAU EDELSTOFF 50cl (5.6%) A brighter, sweeter, more sparkly lager	£4.25
KROMBACHER PILS 50cl (4.8%) Finely bitter taste with a full flavoured aroma	£3.50
JEVER 50cl (4.9%) Bitter & hoppy pilsner	£3.50
SPECIALLY SELECTED WEIZEN BEER Naturally cloudy & fruity	£3.75
BROOKLYN LAGER 35.5cl (5.1%) Caramel malts with a floral hop aroma	£2.95
FLYING DOG SNAKE DOG IPA 35.5cl (7.1%) Citrus hop aroma with caramel malt notes	£4.00
FLYING DOG DOGGIE STYLE 35.5cl (5.5%) Grassy, citrus aromas with a sweet body	£3.50
ODELL CUT THROAT PORTER 35.5cl (4.8%) Smooth & robust	£4.00
ODELL 5 BARREL ALE 35.5cl (5.2%) Fresh lively flavour & aroma	£4.00
ODELL IPA 35.5cl (7%) Distinctive & incredible character	£4.50
CIDER	
GWYNT AUTUMN MAGIC MEDIUM 50cl (4.%) Bursts with an autumnal fruity aroma	£3.75
GWYNT BLACK DRAGON MED/DRY 50cl (6.5%) Award winning	£3.75
GWYNT ORCHARD GOLD MEDIUM 50cl (4.9%) Smooth flavour with a fresh crisp sharpness	£3.50
DE TROCH PEACH25cl (3.5%)	£2.75
DE TROCH STRAWBERRY25cl (3.5%) Belgium fruit beers	£2.75
HOOK NORTON OLD HOOKY50cl (4.6%) Beautifully balanced fruity beer	£3.50
RUDGATE BATTLE AXE 50cl (4.8%) Smooth & robust bitter	£3.25
RUDGATE RUBY MILD 50cl (4.4%) Soft rich complex	£3.25
RUDGATE VIKING 50cl (3.8%) Sweet with honey & pineapple notes	£3.25
NAYLORS PINNACLE BITTER 50cl (3.9%) Bitterness with delicate aromas	£2.95
PERONI RED 33cl (4.7%)	£2.75
SUPER BOCK 33cl (5.6%) Refreshing smooth lagers for all occasions	£2.75

MANY SEASONAL SPECIALS ASK FOR DETAILS

Image ©Restaurant Guru

Model Config

The `model_config` is a dictionary of options that apply to the model.

The `model_config` controls the behaviour of the entire model, and can be used to apply some options in the Field function throughout.

When inheriting a pydantic model, the config is also inherited, with any additional config options merged in.

Model Config

```python
from pydantic import BaseModel, ConfigDict

class User(BaseModel):
    model_config = ConfigDict(str_max_length=10)

    name: str = Field(default='John Doe', min_length=1)
    ...
```

```python
from pydantic import BaseModel

class User(BaseModel, str_max_length=10):
    name: str = Field(default='John Doe', min_length=1)
    ...
```

Model Config

From the full set of config options there are two options I have found particularly useful:

- `validate_assignment`
- `extra`

Validate Assignment

When True, `validate_assignment` revalidates the model when any of the fields are changed (by assignment, NOT by mutation).

This is very useful, and ensures models remain defined as intended throughout their lifetime.

However, this can cause problems . . .

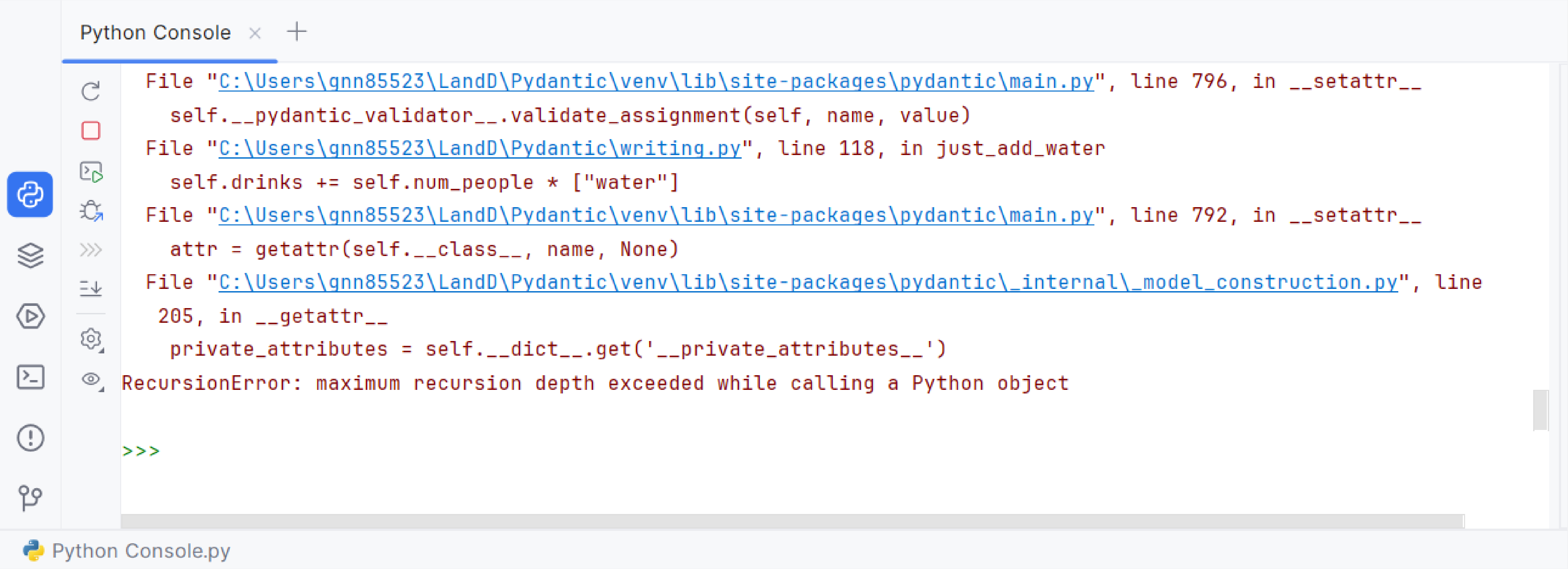
Validate Assignment

```python
from pydantic import BaseModel

class Customer(BaseModel, validate_assignment=True):
    name: str = ""
    drinks: list[str] = []
    num_people: int = 1

    @model_validator(mode='after')
    def just_add_water(self) -> 'Customer':
        """Give each member of the party a complimentary tap water."""
        self.drinks += self.num_people * ["water"]
        return self
```

Validate Assignment



```
  File "C:\Users\gnn85523\LandD\Pydantic\venv\lib\site-packages\pydantic\main.py", line 796, in __setattr__
    self.__pydantic_validator__.validate_assignment(self, name, value)
  File "C:\Users\gnn85523\LandD\Pydantic\writing.py", line 118, in just_add_water
    self.drinks += self.num_people * ["water"]
  File "C:\Users\gnn85523\LandD\Pydantic\venv\lib\site-packages\pydantic\main.py", line 792, in __setattr__
    attr = getattr(self.__class__, name, None)
  File "C:\Users\gnn85523\LandD\Pydantic\venv\lib\site-packages\pydantic\_internal\_model_construction.py", line
 205, in __getattr__
    private_attributes = self.__dict__.get('__private_attributes__')
RecursionError: maximum recursion depth exceeded while calling a Python object

>>>
```

Science and
Technology
Facilities Council

ISIS Neutron and
Muon Source

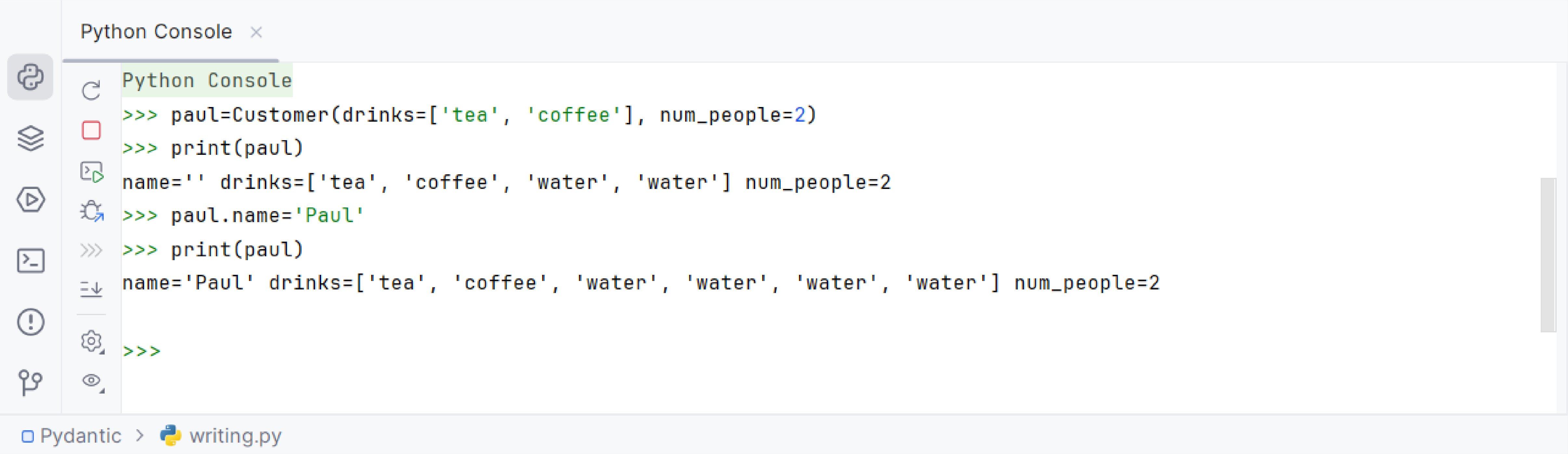
Validate Assignment

```python
from pydantic import BaseModel

class Customer(BaseModel, validate_assignment=True):
    name: str = ""
    drinks: list[str] = []
    num_people: int = 1

    @model_validator(mode='after')
    def just_add_water(self) -> 'Customer':
        """Give each member of the party a complimentary tap water."""
        self.drinks.extend(self.num_people * ["water"])
        return self
```

Validate Assignment



```
Python Console
>>> paul=Customer(drinks=['tea', 'coffee'], num_people=2)
>>> print(paul)
name='' drinks=['tea', 'coffee', 'water', 'water'] num_people=2
>>> paul.name='Paul'
>>> print(paul)
name='Paul' drinks=['tea', 'coffee', 'water', 'water', 'water', 'water'] num_people=2

>>>
```

Validate Assignment

To avoid these problems, we need to think carefully about what we do inside validators.

One solution is to make changes by mutation rather than assignment where possible.

We can also use the `model_post_init` routine to modify the model prior to model validation.

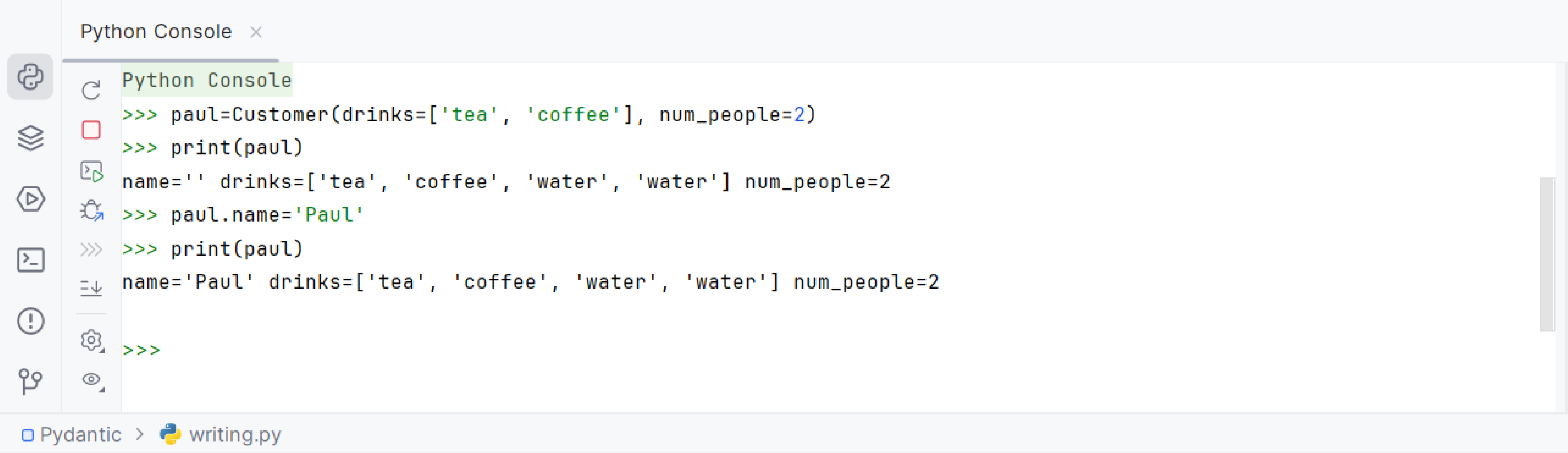
Model Post Init

```python
from pydantic import BaseModel
from typing import Any

class Customer(BaseModel, validate_assignment=True):
    name: str = ""
    drinks: list[str] = []
    num_people: int = 1

    def model_post_init(self, __context: Any) -> None:
        """Give each member of the party a complimentary tap water."""
        self.drinks.extend(self.num_people * ["water"])
```

Model Post Init



```
Python Console
>>> paul=Customer(drinks=['tea', 'coffee'], num_people=2)
>>> print(paul)
name='' drinks=['tea', 'coffee', 'water', 'water'] num_people=2
>>> paul.name='Paul'
>>> print(paul)
name='Paul' drinks=['tea', 'coffee', 'water', 'water'] num_people=2


>>>
```

Extra Fields

What happens when we include undefined fields in the model initialisation?

It depends on the value of `extra` in the `model_config`.

- `allow` – Include any extra attributes in the model
- `forbid` – Forbid any extra attributes, raise an error if any are included
- `ignore` – Ignore any extra attributes (default)

If `extra=“allow”`, then the extra fields are listed in the `model_extra` attribute of the model.

Extra Fields

```
from pydantic import BaseModel, ConfigDict

class User(BaseModel):
    model_config = ConfigDict(extra='ignore')

    name: str


user = User(name='John Doe', age=20)
print(user)
#> name='John Doe'
```

Extra Fields

```python
from pydantic import BaseModel, ConfigDict

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

    name: str


user = User(name='John Doe', age=20)
print(user)
#> name='John Doe' age=20
```

Extra Fields

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

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

    name: str

try:
    User(name='John Doe', age=20)
except ValidationError as e:
    print(e)
    '''
    1 validation error for User
    age
    Extra inputs are not permitted [type=extra_forbidden, input_value=20, input_type=int]
    '''
```

Free Play

- Try using the `model_config` – set `validate_assignment` to True and see what happens as you change the fields. How should you set `extra`?

- Include a model for Tea alongside Coffee – think about an appropriate inheritance structure.

- Include hot chocolate – make a list of toppings with some validation.

- The methods of making coffee will themselves have options to choose from – try splitting them out into individual models and selecting them from the Coffee model using a discriminated union.

Free Play

- Try strict mode. What happens when you include `strict=True` in the `model_config`? If you set `strict=False` in the Field function for one of the fields, what happens then?

- Add a water field to the Coffee model, using the pint library to ensure a particular volume of water is specified. What validators are needed here?

  - You'll need to include `arbitrary_types_allowed=True` in the `model_config` here.

- . . . And anything else you can come up with!

Links

- Pydantic: https://docs.pydantic.dev/latest/
- ISIS: https://www.isis.stfc.ac.uk/Pages/home.aspx
- Typing module: https://docs.python.org/3/library/typing.html
- Pint: https://pint.readthedocs.io/en/stable/

- Migration Guide: https://docs.pydantic.dev/2.0/migration/
- Coercion: https://docs.pydantic.dev/latest/api/standard_library_types/
- Fields: https://docs.pydantic.dev/latest/concepts/fields/
- Validators: https://docs.pydantic.dev/latest/concepts/validators/
- Model Config: https://docs.pydantic.dev/latest/api/config/