Instructions

- Open a terminal and type:

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

- Activate the virtual environment (see: https://docs.python.org/3/library/venv.html)

- `pip install -r requirements.txt`
- Open your favourite editor and get ready to code!

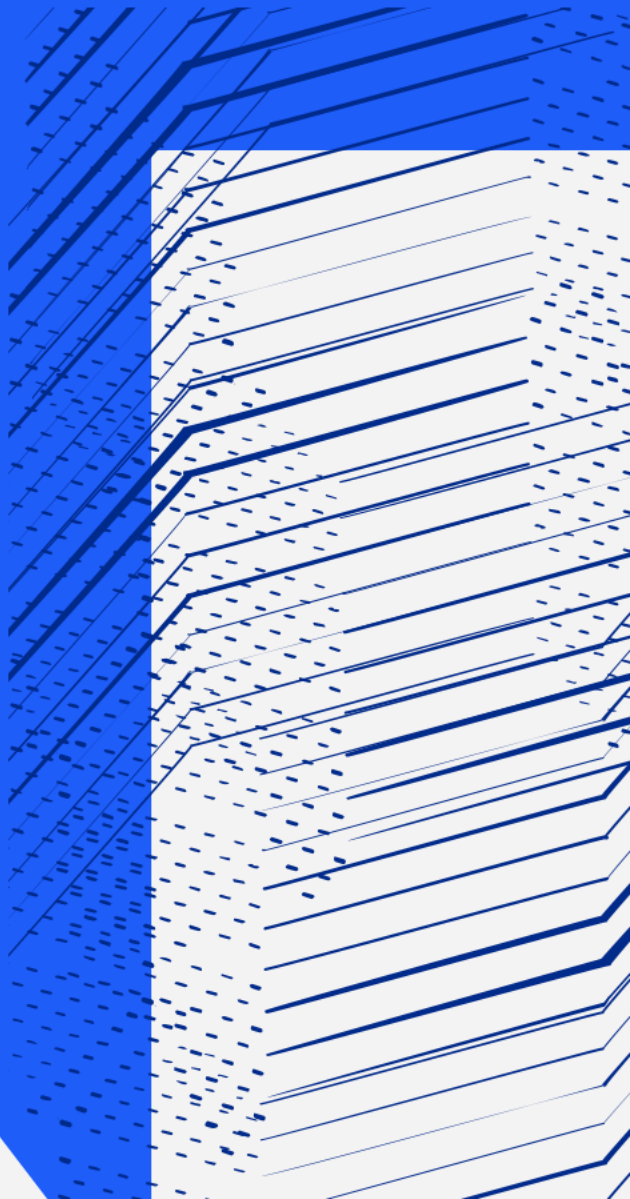
Science and Technology Facilities Council

ISIS Neutron and Muon Source

Pydantic: A Package for Picky Python Programmers

Paul Sharp
ISIS Research Software Engineering Team
December 2023

@DrPaulSharp



About Me

- PhD at University of York 2011-2015.
- Postdoc at University of Liverpool, 2016-2020.
- Worked at STFC since 2020, first with Scientific Computing & SuperSTEM . . .
- . . . and now as a RSE at ISIS Neutron & Muon Source.

Codes:

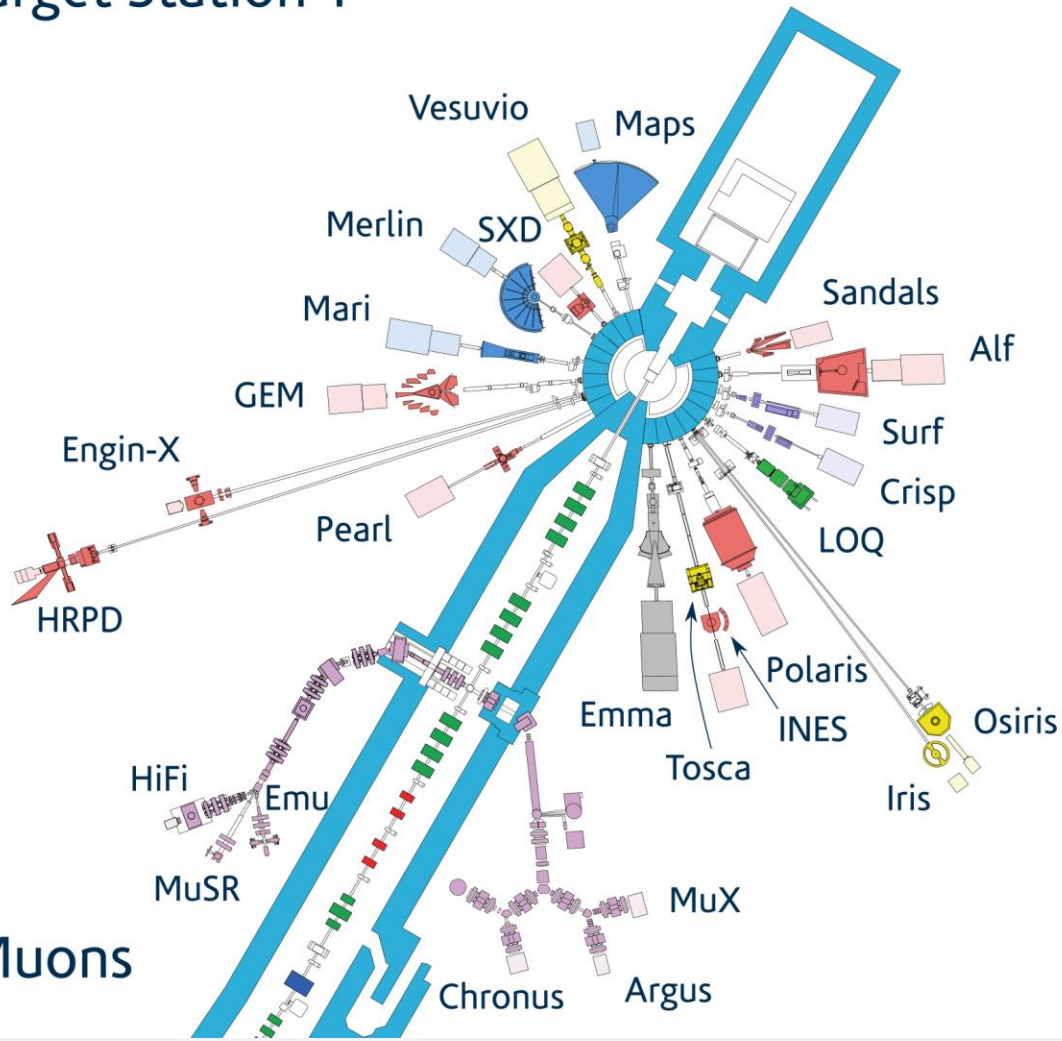
ChemDASH, InverSTEM, RAT

Science and Technology Facilities Council

ISIS Neutron and Muon Source

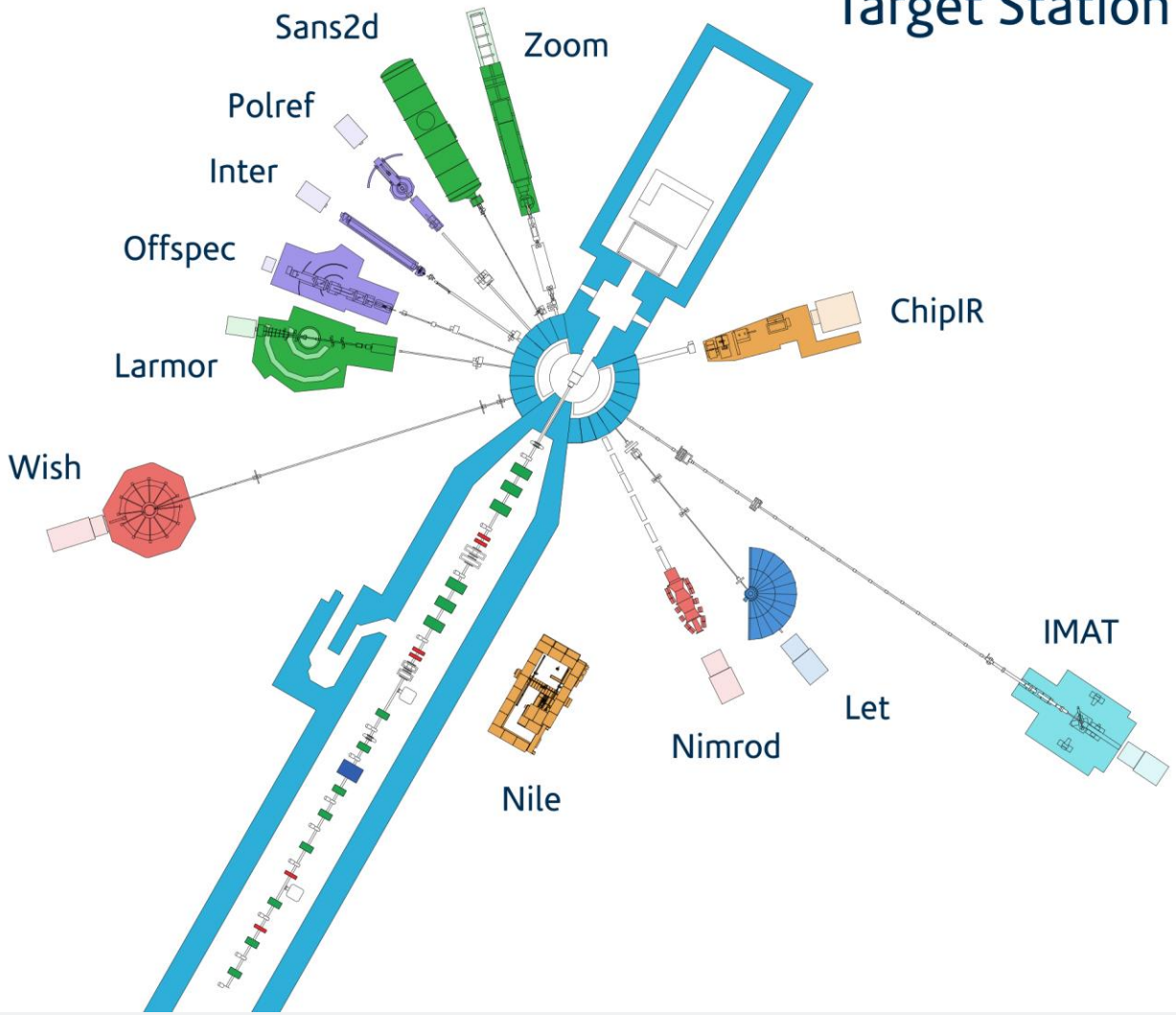

UKRI
Science and Technology Facilities Council
ISIS Neutron and Muon Source

Target Station 1



Muons

Target Station 2



Science and Technology Facilities Council

ISIS Neutron and Muon Source

Research Software Engineering

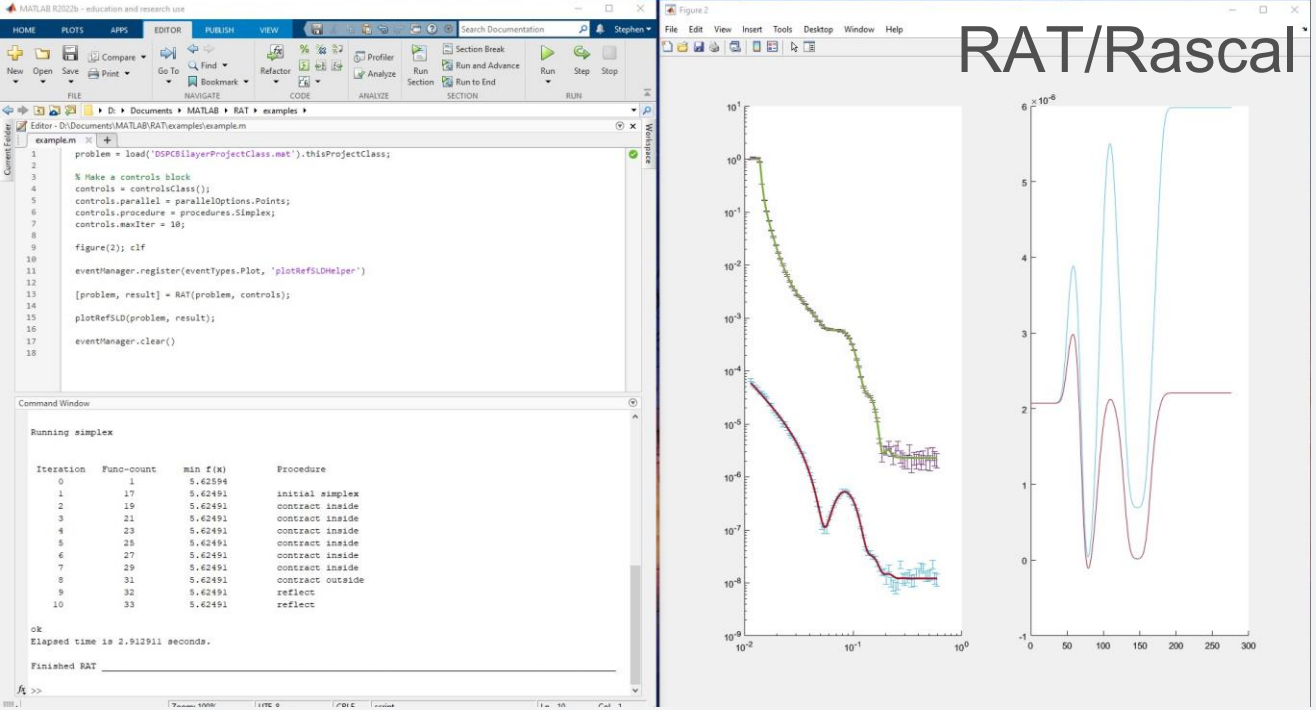
The term “Research Software Engineer” (RSE) was first coined in 2012.

This was done with the aim of promoting the role of software developers in scientific research and to establish a viable career path.

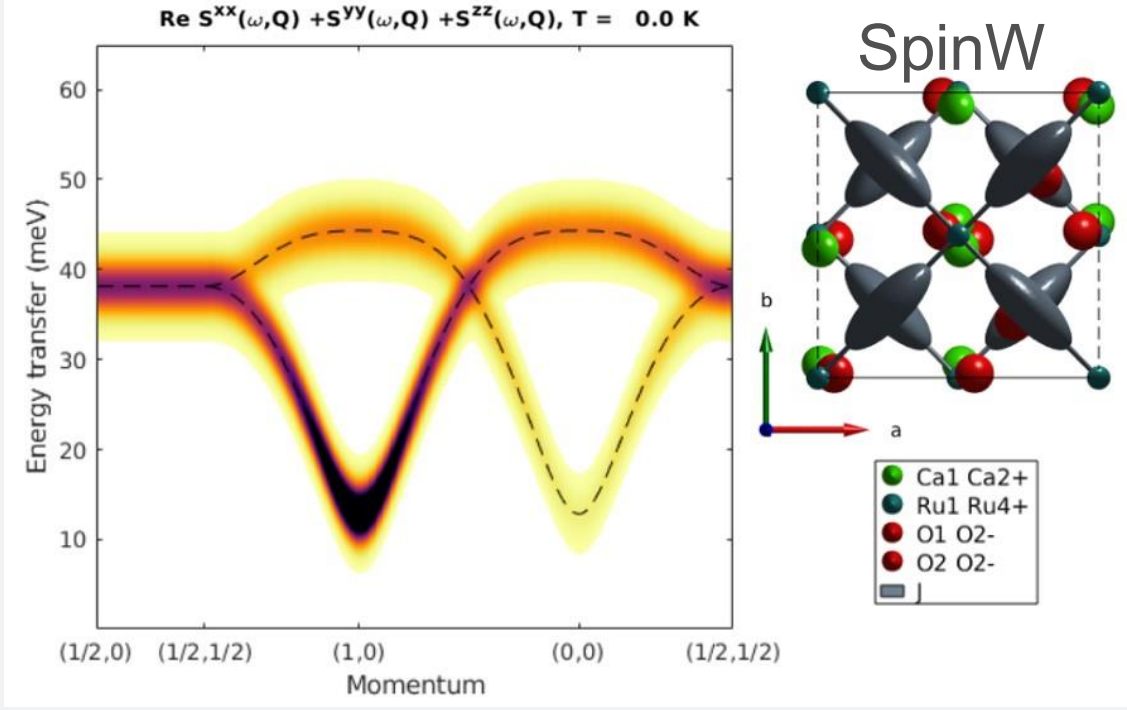
First RSE workshop in 2013, where the RSE association was established.
First RSE conference in 2016.

RSE groups have been established in academic institutions and large-scale facilities.

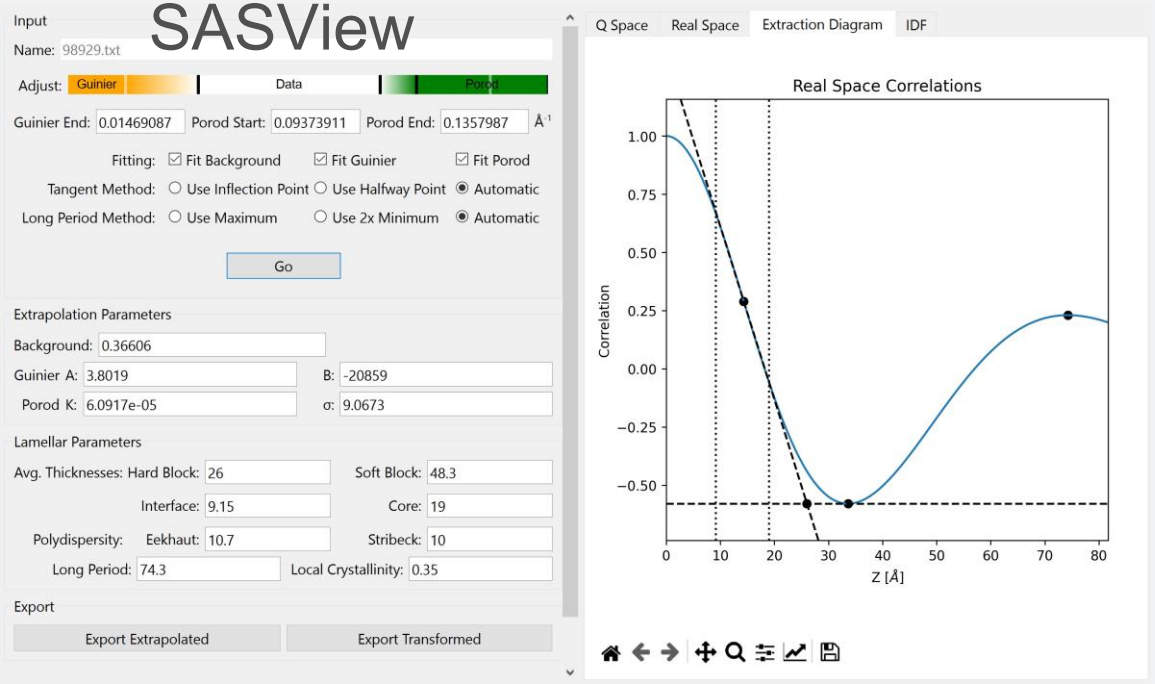
RAT/Rascal



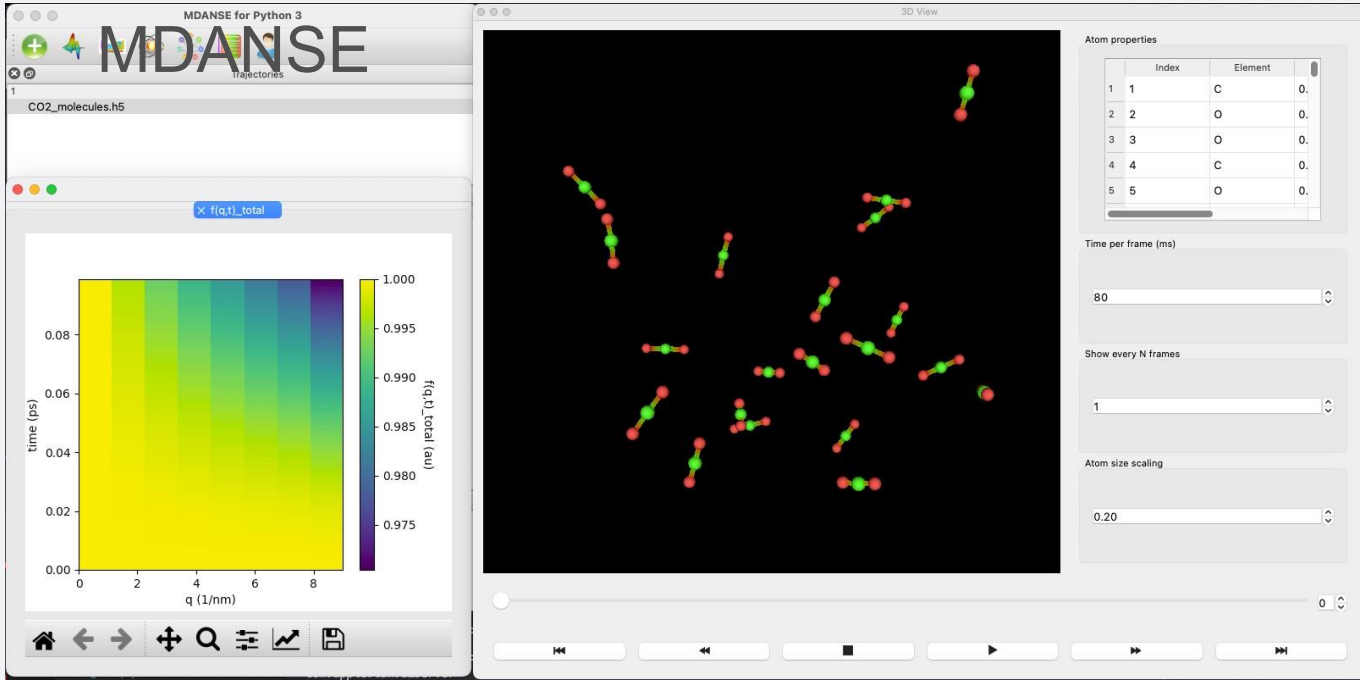
SpinW

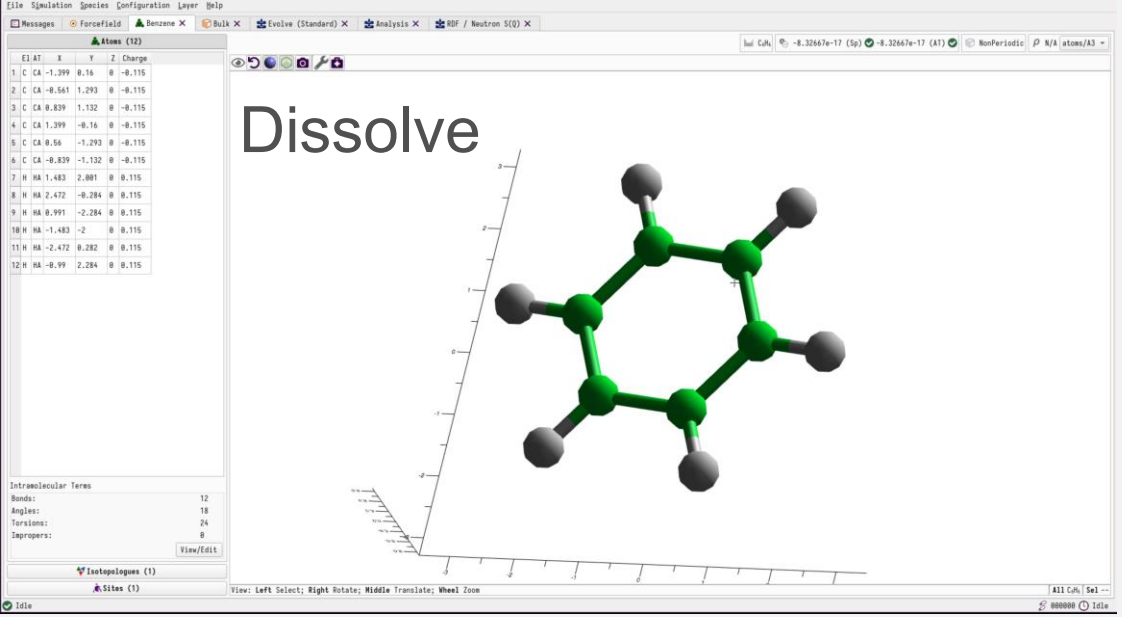


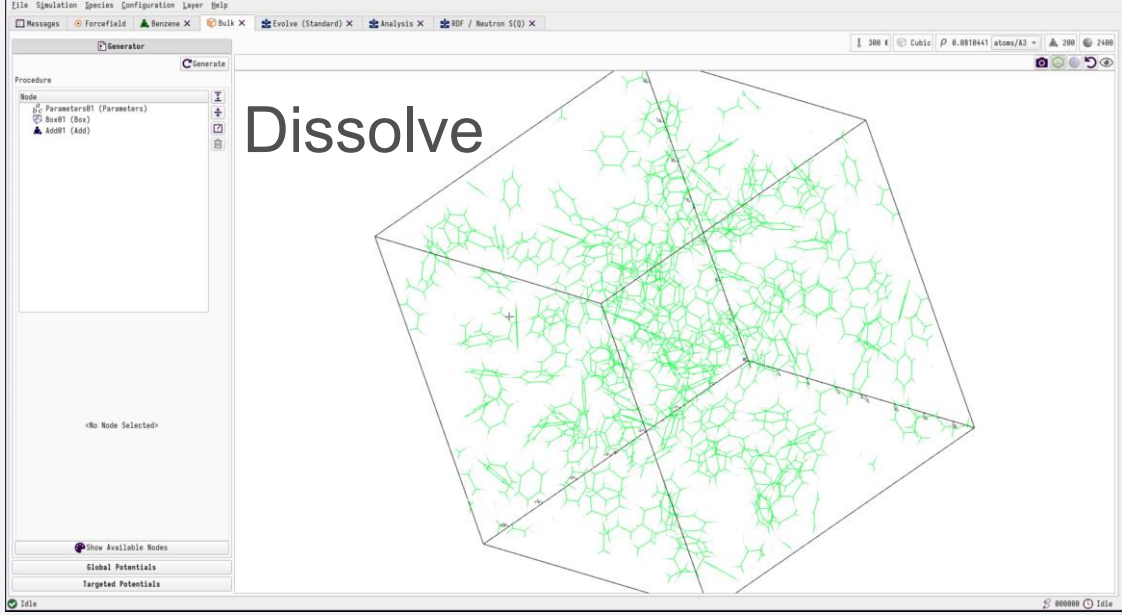
SASView

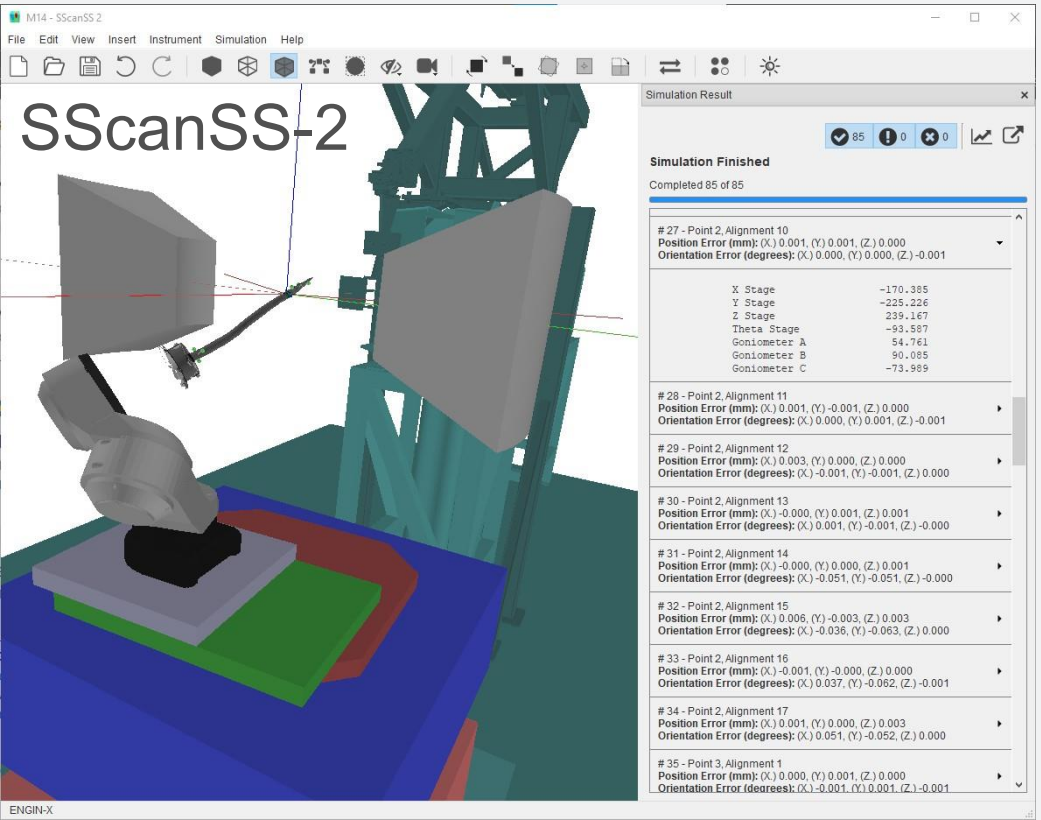


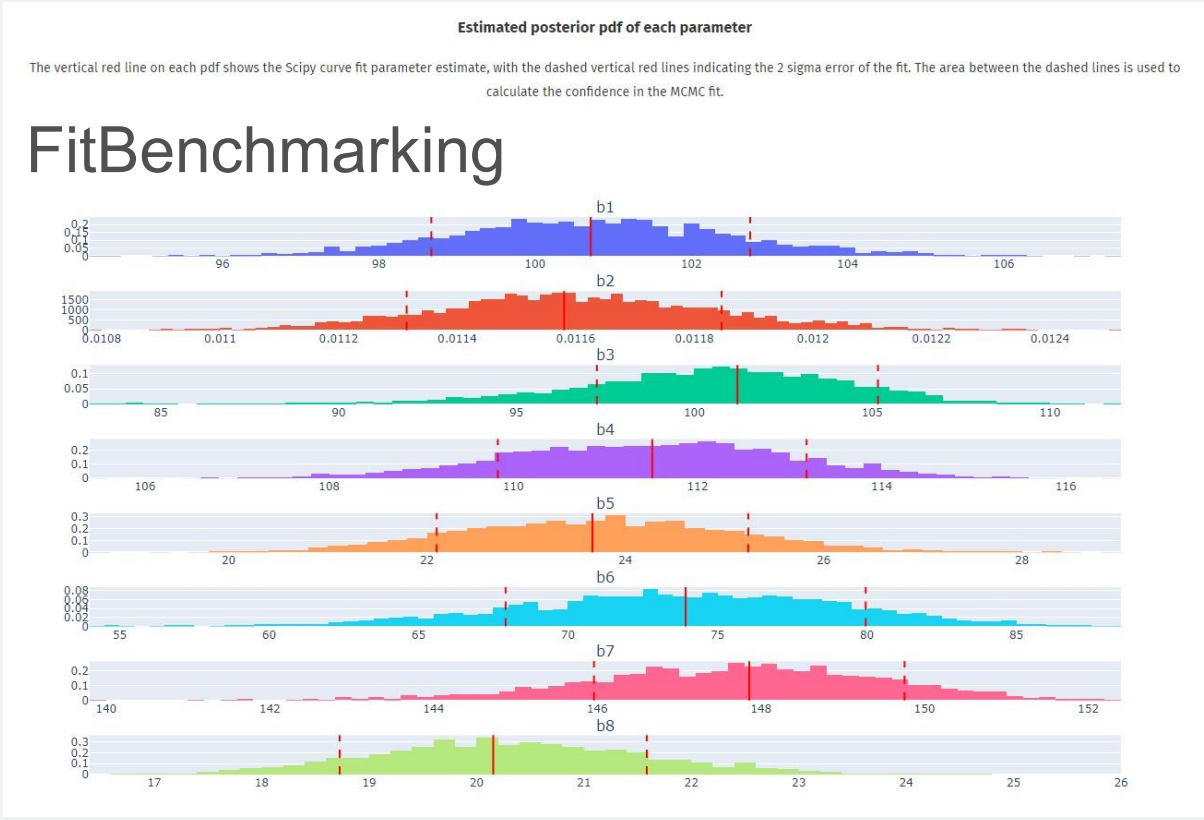
MDANSE




Dissolve


Dissolve


SScanSS-2
Simulation Result
Simulation Finished
Completed 85 of 85


Estimated posterior pdf of each parameter
The vertical red line on each pdf shows the Scipy curve fit parameter estimate, with the dashed vertical red lines indicating the 2 sigma error of the fit. The area between the dashed lines is used to calculate the confidence in the MCMC fit.
FitBenchmarking
b1
b2
b3
b4
b5
b6
b7
b8

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 123M times a month, and used by some of the largest and most recognisable organisations in the world, including:

Science and Technology Facilities Council

ISIS Neutron and Muon Source

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: https://docs.pydantic.dev/2.0/migration/

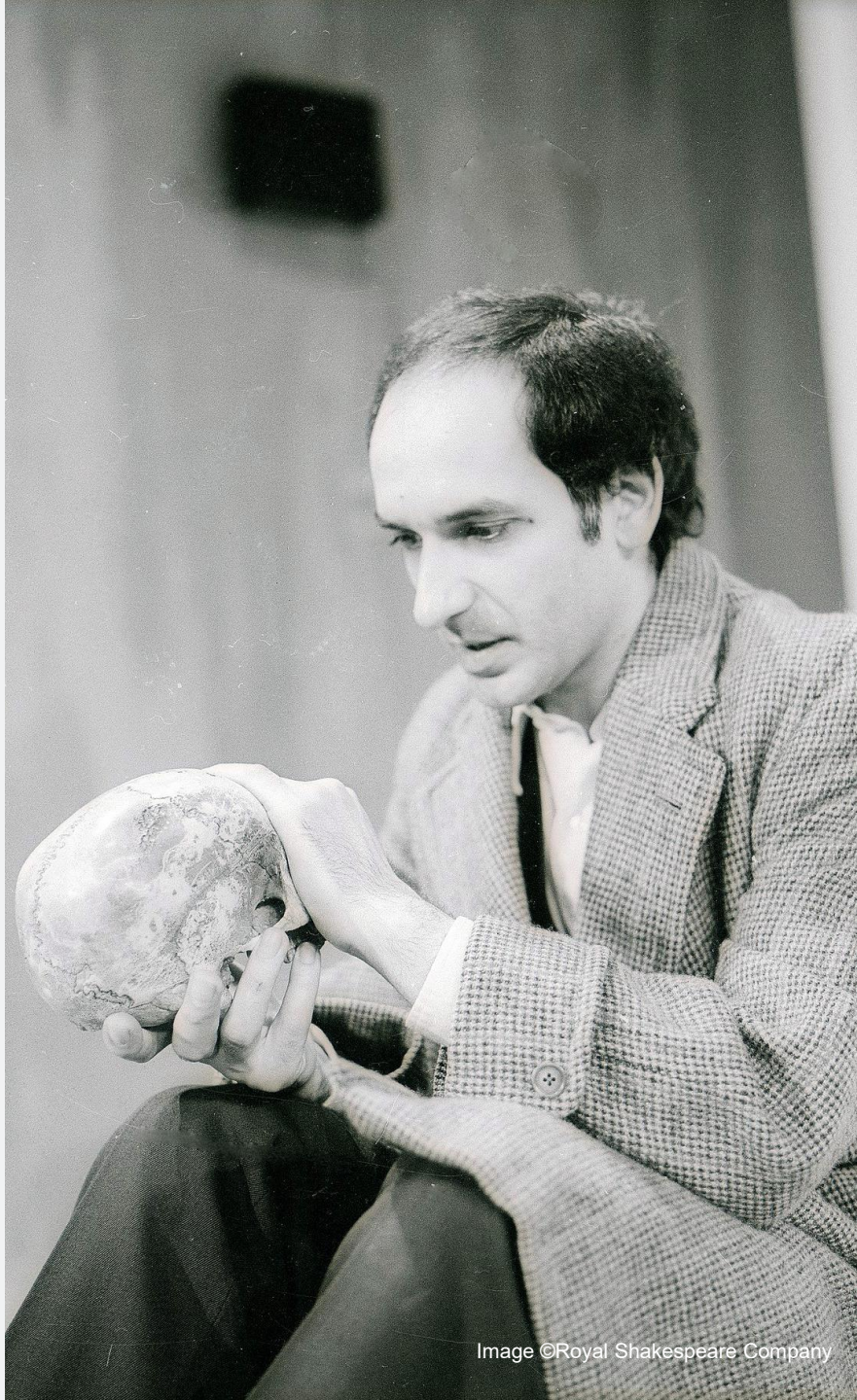


Image ©Royal Shakespeare Company

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 https://peps.python.org/pep-0484/ 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.

```
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 https://mypy-lang.org/

PEP 484 also introduces the typing module https://docs.python.org/3/library/typing.html, 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 https://peps.python.org/pep-0557/.

```
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

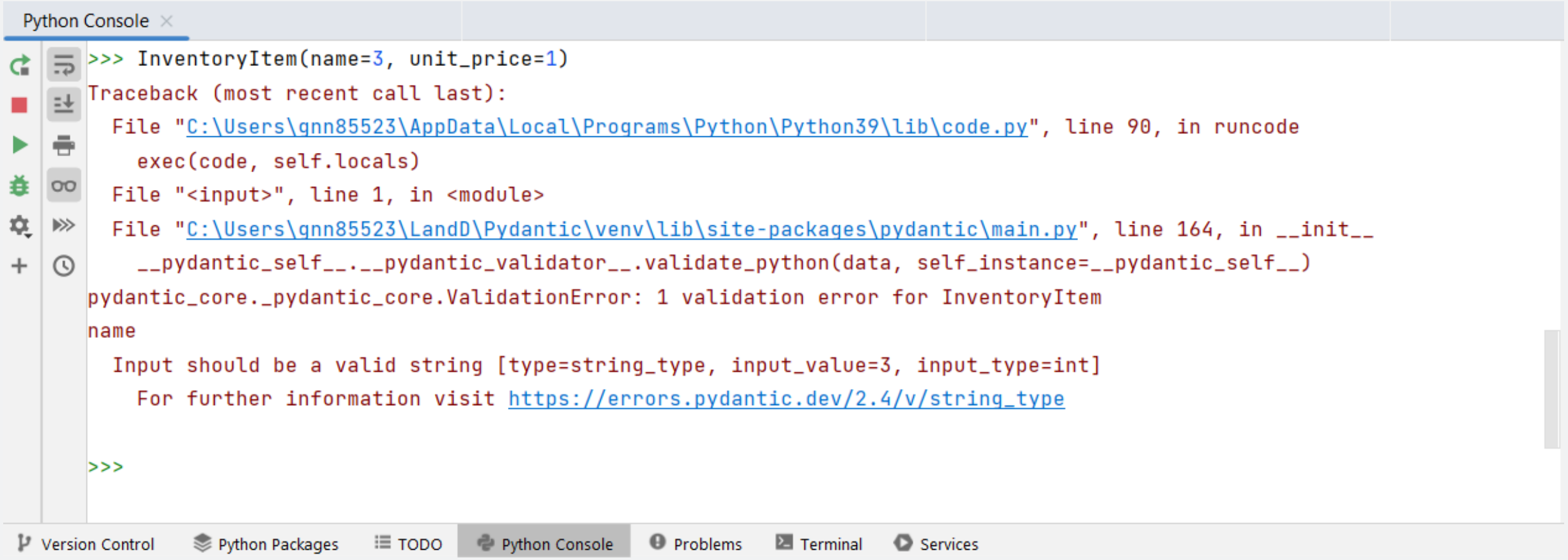
With pydantic, we can enforce our type hints. When a class is defined that inherits from the pydantic `BaseModel`, an error is raised if one of the fields is of the incorrect type.

```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



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 . . .

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.

Image ©Restaurant Guru

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?

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. https://docs.pydantic.dev/latest/api/standard_library_types/
- 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.

Restricting Strings

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

```
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"
    ...
```

Restricting Strings

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.
https://docs.pydantic.dev/latest/concepts/fields/

The basic syntax is:

```
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 one or more 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).

A field validator can be used on one or more fields.

Field Validators

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

```
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: https://peps.python.org/pep-0593/

Annotated Validators

```python
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.

Science and Technology Facilities Council

ISIS Neutron and Muon Source

Coffee Size

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 . . .

Image ©Restaurant Guru

Science and Technology Facilities Council

ISIS Neutron and Muon Source

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

The config options are listed here: https://docs.pydantic.dev/latest/api/config/

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 . . .

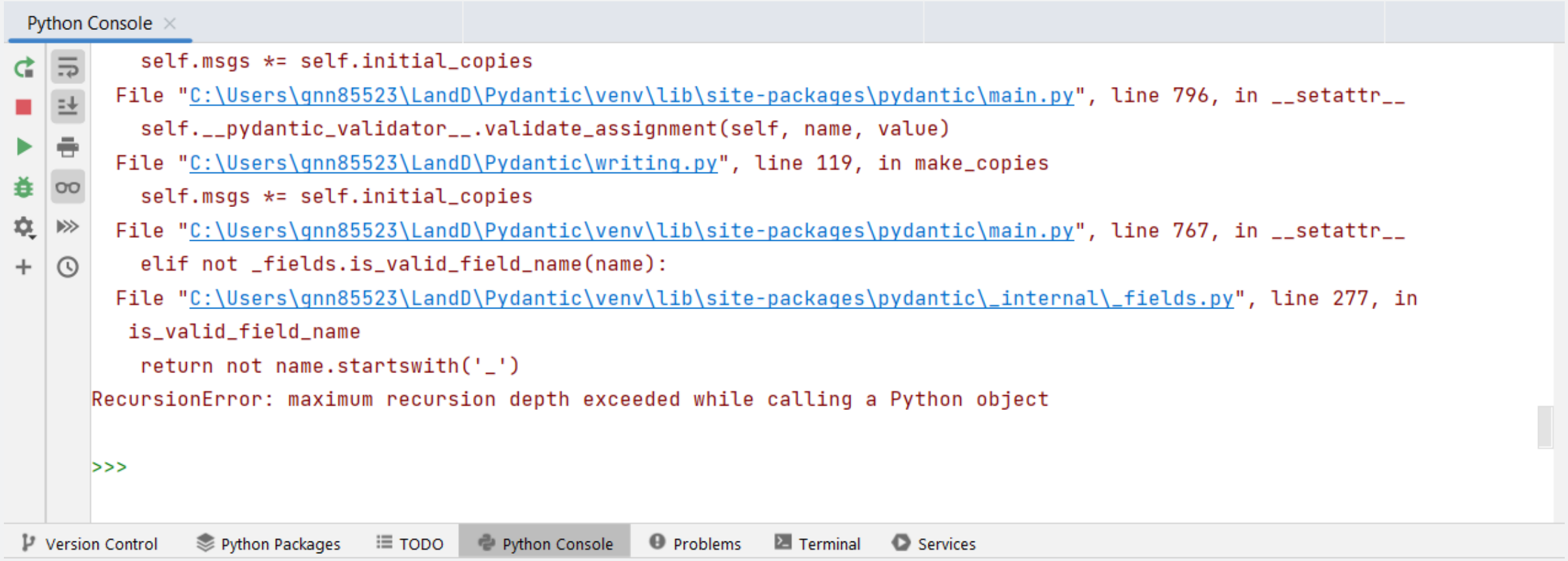
Validate Assignment

```python
from pydantic import BaseModel

class User(BaseModel, validate_assignment=True):
    name: str = ""
    msgs: list[str] = []
    initial_copies: int = 2

    @model_validator(mode='after')
    def make_copies(self) -> 'User':
        self.msg *= self.initial_copies
        return self
```

Validate Assignment



Science and Technology Facilities Council

ISIS Neutron and Muon Source

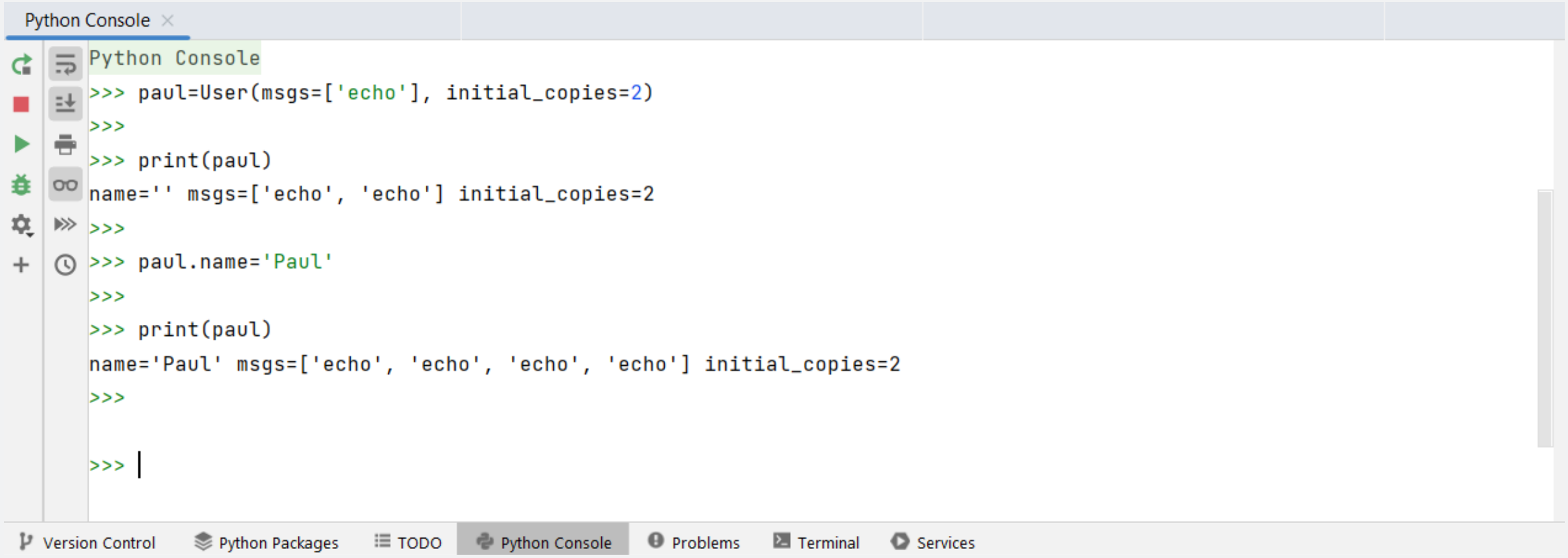
Validate Assignment

```python
from pydantic import BaseModel

class User(BaseModel, validate_assignment=True):
    name: str = ""
    msgs: list[str] = []
    initial_copies: int = 2

    @model_validator(mode='after')
    def make_copies(self) -> 'User':
        orig_list = self.msgs
        for i in range(1, self.initial_copies):
            self.msgs.extend(orig_list)
        return self
```

Validate Assignment



Science and Technology Facilities Council

ISIS Neutron and Muon Source

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.

https://docs.pydantic.dev/latest/api/base_model/#pydantic.main.BaseModel.model_post_init

Model Post Init

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

class User(BaseModel, validate_assignment=True):
    name: str = ""
    msgs: list[str] = []
    initial_copies: int = 2

    def model_post_init(self, __context: Any) -> None:
        orig_list = self.msgs
        for i in range(1, self.initial_copies):
            self.msgs.extend(orig_list)
```

Model Post Init

Science and Technology Facilities Council

ISIS Neutron and Muon Source

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

```python
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** https://docs.pydantic.dev/latest/concepts/unions/#discriminated-unions.

Free Play

- Add a water field to the Coffee model, using the pint library (https://pint.readthedocs.io/en/stable/) 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/