Contents

[dispatcher-style design 3](#_Toc199524123)

[✅ Goal 3](#_Toc199524124)

[✅ Implementation 4](#_Toc199524125)

[✅ Usage Examples 4](#_Toc199524126)

[✅ Advantages 4](#_Toc199524127)

[🛠 Optional: Use Dispatch Dictionary for Cleaner Design 5](#_Toc199524128)

[dynamic parameter validation systembased onmultiple selector inputs 6](#_Toc199524129)

[✅ Flexible Design Pattern 6](#_Toc199524130)

[🧱 Step-by-Step Implementation 7](#_Toc199524131)

[✅ Example Usage 8](#_Toc199524132)

[✅ Benefits 8](#_Toc199524133)

[type-aware, object-level validation 9](#_Toc199524134)

[a modular, scalable architecture for this sake 13](#_Toc199524135)

[✅ Architectural Pattern for This Use Case 13](#_Toc199524136)

[🔧 HOW TO ORGANIZE IT 13](#_Toc199524137)

[1. Modular Class Definitions 13](#_Toc199524138)

[2. Place Config in a Dedicated Module 14](#_Toc199524139)

[3. Generic Builder/Validator for Any Module 14](#_Toc199524140)

[4. Build FRSMOTE with This 15](#_Toc199524141)

[✅ Example Usage 15](#_Toc199524142)

[✅ Advantages of This Architecture 16](#_Toc199524143)

[📦 Optional Enhancements 16](#_Toc199524144)

[PYDANTIC 16](#_Toc199524145)

[✅ Why pydantic 17](#_Toc199524146)

[🧱 Step-by-Step with pydantic 17](#_Toc199524147)

[1. Install Pydantic 17](#_Toc199524148)

[2. Define Component Models 17](#_Toc199524149)

[3. Create a Union Model for fr\_model Selection 18](#_Toc199524150)

[4. HGF Config Models (Optional) 18](#_Toc199524151)

[5. Generic Factory Function 18](#_Toc199524152)

[6. Main Class (e.g. FRSMOTE) with Nested Validation 18](#_Toc199524153)

[✅ Example Usage 19](#_Toc199524154)

[🧪 Output 19](#_Toc199524155)

[✅ Advantages 19](#_Toc199524156)

[Optional: Use attrs Instead? 20](#_Toc199524157)

# dispatcher-style design

I have a function in python having a string parameter fr\_model as its input. This is the name of a model we can select. Each model has different set of parameters with different values. I have another parameter like this called hgf which is a string like before and with any selection, needs to get some float values. how can I implement that function with flexibility?

let me describe more. If user inputs fr\_model='ITFRS', then it needs a string parameter which could be any of{'tnorm1', 'tnorm2', 'tnorm3'}. if user inputs fr\_model='VQRS', then the function needs to get alpha\_lower, beta\_lower as well with it that are float values. How can we implement this function?

Thank you — that's much clearer now!

You're describing a function where the **required additional parameters vary depending on the value of fr\_model**. This is a great use case for a **dispatcher-style** design or for validating **conditional arguments** based on the model type.

### ✅ Goal

Build a function like:

python

CopyEdit

def run\_model(fr\_model: str, \*\*kwargs):

...

Which dynamically:

* Accepts fr\_model (e.g. 'ITFRS', 'VQRS')
* Depending on fr\_model, validates and extracts required parameters (like 'tnorm1', alpha\_lower, beta\_lower, etc.)
* Raises helpful errors if required parameters are missing or invalid.

### ✅ Implementation

python

CopyEdit

def run\_model(fr\_model: str, \*\*kwargs):

if fr\_model == "ITFRS":

tnorm = kwargs.get("tnorm")

if tnorm not in {"tnorm1", "tnorm2", "tnorm3"}:

raise ValueError("For ITFRS, 'tnorm' must be one of {'tnorm1', 'tnorm2', 'tnorm3'}")

return f"Running ITFRS with tnorm: {tnorm}"

elif fr\_model == "VQRS":

try:

alpha = float(kwargs["alpha\_lower"])

beta = float(kwargs["beta\_lower"])

except KeyError as e:

raise ValueError(f"Missing required parameter for VQRS: {e.args[0]}")

except ValueError:

raise ValueError("alpha\_lower and beta\_lower must be float values")

return f"Running VQRS with alpha\_lower: {alpha}, beta\_lower: {beta}"

else:

raise ValueError(f"Unsupported fr\_model: {fr\_model}")

### ✅ Usage Examples

python

CopyEdit

print(run\_model("ITFRS", tnorm="tnorm2"))

# Output: Running ITFRS with tnorm: tnorm2

print(run\_model("VQRS", alpha\_lower=0.1, beta\_lower=0.5))

# Output: Running VQRS with alpha\_lower: 0.1, beta\_lower: 0.5

### ✅ Advantages

* **Flexible**: You can support any number of models with unique parameter sets.
* **Extensible**: Just add another elif fr\_model == ... branch.
* **Safe**: Validates required parameters clearly.

### 🛠 Optional: Use Dispatch Dictionary for Cleaner Design

If this grows, use a dispatch map to separate logic:

python

CopyEdit

def run\_itfrs(\*\*kwargs):

tnorm = kwargs.get("tnorm")

if tnorm not in {"tnorm1", "tnorm2", "tnorm3"}:

raise ValueError("Invalid tnorm")

return f"ITFRS with {tnorm}"

def run\_vqrs(\*\*kwargs):

alpha = float(kwargs["alpha\_lower"])

beta = float(kwargs["beta\_lower"])

return f"VQRS with {alpha}, {beta}"

model\_dispatch = {

"ITFRS": run\_itfrs,

"VQRS": run\_vqrs,

}

def run\_model(fr\_model: str, \*\*kwargs):

if fr\_model not in model\_dispatch:

raise ValueError(f"Unsupported fr\_model: {fr\_model}")

return model\_dispatch[fr\_model](\*\*kwargs)

Let me know if you’d like it to support default values or optional parameters too.

# **dynamic parameter validation system** based on **multiple selector inputs**

Yes, I need support default values or optional parameters too. Moreover, what if I have more input parameters like that which based on selection, need different parameters?

###### ChatGPT said:

Perfect — you're describing a **dynamic parameter validation system** based on **multiple selector inputs** (like fr\_model, hgf, etc.), where:

1. Each selector (e.g., fr\_model, hgf) determines:
   * Required parameters
   * Optional parameters (with defaults)
2. The function should:
   * Validate these based on user input
   * Raise errors for missing required parameters
   * Apply defaults where appropriate

### ✅ Flexible Design Pattern

We can define a **configuration map** for each selector (e.g., fr\_model, hgf) that specifies:

* Required parameters (with type or valid values)
* Optional parameters (with default values)

### 🧱 Step-by-Step Implementation

#### Step 1: Define Config

python

CopyEdit

model\_config = {

"fr\_model": {

"ITFRS": {

"required": {"tnorm": {"type": str, "choices": {"tnorm1", "tnorm2", "tnorm3"}}},

"optional": {"verbose": {"type": bool, "default": False}},

},

"VQRS": {

"required": {

"alpha\_lower": {"type": float},

"beta\_lower": {"type": float}

},

"optional": {"gamma": {"type": float, "default": 0.5}},

},

},

"hgf": {

"HGF1": {

"required": {"learning\_rate": {"type": float}},

"optional": {"momentum": {"type": float, "default": 0.9}},

},

"HGF2": {

"required": {},

"optional": {"decay": {"type": float, "default": 0.01}},

},

}

}

#### Step 2: Validation & Dispatch

python

CopyEdit

def validate\_params(selection: str, config: dict, \*\*kwargs):

if selection not in config:

raise ValueError(f"Invalid selection: {selection}. Must be one of {list(config)}")

conf = config[selection]

validated = {}

# Validate required

for key, spec in conf.get("required", {}).items():

if key not in kwargs:

raise ValueError(f"Missing required parameter '{key}' for selection '{selection}'")

val = kwargs[key]

if "choices" in spec and val not in spec["choices"]:

raise ValueError(f"Invalid value for '{key}': {val}. Must be one of {spec['choices']}")

validated[key] = spec["type"](val)

# Validate optional

for key, spec in conf.get("optional", {}).items():

val = kwargs.get(key, spec["default"])

validated[key] = spec["type"](val)

return validated

#### Step 3: Main Function

python

CopyEdit

def run\_model(fr\_model: str, hgf: str, \*\*kwargs):

fr\_model\_params = validate\_params(fr\_model, model\_config["fr\_model"], \*\*kwargs)

hgf\_params = validate\_params(hgf, model\_config["hgf"], \*\*kwargs)

# Do something with validated parameters

print(f"Running {fr\_model} with: {fr\_model\_params}")

print(f"Using HGF model {hgf} with: {hgf\_params}")

return fr\_model\_params, hgf\_params

### ✅ Example Usage

python

CopyEdit

run\_model(

fr\_model="VQRS",

hgf="HGF1",

alpha\_lower=0.1,

beta\_lower=0.3,

learning\_rate=0.01

)

python

CopyEdit

run\_model(

fr\_model="ITFRS",

hgf="HGF2",

tnorm="tnorm2"

)

### ✅ Benefits

* Scales to **any number of selectors** (add more like hgf, optimizer, etc.)
* Easy to **define/adjust logic in config** without hardcoding
* Supports **defaults, types, and allowed choices**
* Centralized, **clean validation**

Would you like me to turn this into a class-based API, or make it work with external JSON configs for real-world deployment?

# type-aware, object-level validation

What if in one of the model selections, like ITFRS, we need to get an object which is of custom type custome\_class?

**ChatGPT said:**

Great question — now we're stepping into **type-aware, object-level validation**, where some parameters aren't simple types like float or str, but actual instances of a **custom class** (e.g., CustomClass).

This is fully compatible with the configuration-driven approach. You just need to:

**✅ Step 1: Define Your Custom Class**

python

CopyEdit

class CustomClass:

def \_\_init\_\_(self, name: str):

self.name = name

def \_\_repr\_\_(self):

return f"CustomClass(name={self.name})"

**✅ Step 2: Update Config to Expect That Type**

Here’s how the ITFRS model might expect an instance of CustomClass:

python

CopyEdit

model\_config = {

"fr\_model": {

"ITFRS": {

"required": {

"tnorm": {"type": str, "choices": {"tnorm1", "tnorm2", "tnorm3"}},

"custom\_obj": {"type": CustomClass} # <== custom class expected

},

"optional": {"verbose": {"type": bool, "default": False}},

},

...

},

...

}

**✅ Step 3: Adjust validate\_params() to Allow Class Instance Checking**

Update the validation logic to **allow object-type checking** (don't coerce, just verify isinstance()):

python

CopyEdit

def validate\_params(selection: str, config: dict, \*\*kwargs):

if selection not in config:

raise ValueError(f"Invalid selection: {selection}. Must be one of {list(config)}")

conf = config[selection]

validated = {}

for key, spec in conf.get("required", {}).items():

if key not in kwargs:

raise ValueError(f"Missing required parameter '{key}' for selection '{selection}'")

val = kwargs[key]

# Check for choice restriction

if "choices" in spec and val not in spec["choices"]:

raise ValueError(f"Invalid value for '{key}': {val}. Must be one of {spec['choices']}")

expected\_type = spec["type"]

# For custom class or objects, use isinstance, not casting

if not isinstance(val, expected\_type):

raise TypeError(f"Parameter '{key}' must be of type {expected\_type.\_\_name\_\_}, got {type(val).\_\_name\_\_}")

validated[key] = val

for key, spec in conf.get("optional", {}).items():

val = kwargs.get(key, spec["default"])

expected\_type = spec["type"]

if not isinstance(val, expected\_type):

try:

val = expected\_type(val) # try to coerce if not object

except Exception:

raise TypeError(f"Optional parameter '{key}' must be {expected\_type.\_\_name\_\_}")

validated[key] = val

return validated

**✅ Example Usage**

python

CopyEdit

my\_obj = CustomClass("example")

params = run\_model(

fr\_model="ITFRS",

hgf="HGF2",

tnorm="tnorm2",

custom\_obj=my\_obj

)

**🧠 Recap**

| **Feature** | **How It's Handled** |
| --- | --- |
| Custom types | Use isinstance(val, CustomClass) |
| Coercion for simple types | float(val) etc. |
| Optional parameters | Set defaults via config |
| Choice validation | Use choices in config |

Would you like this turned into a class-based framework or a plugin-based registry for real extension?

# a **modular, scalable architecture for this sake**

What if I want to use the same for many classes? Assume FRSMOTE class uses ITFRS, VQRS, OWAFRS, tnorms etc. ITFRS itself uses parameters which can be changed based on different selection and nedd different sub parameters. Where and how should I implement this config dictionaray?

###### ChatGPT said:

You're now describing a **modular, scalable architecture** where:

1. You have multiple **main algorithms or classes** (e.g., FRSMOTE).
2. These use **pluggable submodules** (like ITFRS, VQRS, OWAFRS, tnorm, etc.).
3. Each module has **its own conditional parameters**, some of which depend on **nested selections**.

## ✅ Architectural Pattern for This Use Case

This is a perfect case for combining:

* **Class-based modular design**
* **Centralized, hierarchical config validation**
* **Flexible dependency injection of submodules**
* **Factory or registry pattern**

## 🔧 HOW TO ORGANIZE IT

### 1. **Modular Class Definitions**

Each component (e.g., ITFRS, VQRS, FRSMOTE) is a class.

python

CopyEdit

class ITFRS:

def \_\_init\_\_(self, tnorm: str, custom\_obj=None):

self.tnorm = tnorm

self.custom\_obj = custom\_obj

python

CopyEdit

class VQRS:

def \_\_init\_\_(self, alpha\_lower: float, beta\_lower: float, gamma: float = 0.5):

...

python

CopyEdit

class FRSMOTE:

def \_\_init\_\_(self, fr\_model: str, hgf: str, \*\*kwargs):

self.fr\_model = build\_fr\_model(fr\_model, \*\*kwargs)

self.hgf\_model = build\_hgf\_model(hgf, \*\*kwargs)

### 2. **Place Config in a Dedicated Module**

Create a file, say: model\_configs.py

python

CopyEdit

from your\_classes import ITFRS, VQRS, CustomClass

model\_config = {

"fr\_model": {

"ITFRS": {

"class": ITFRS,

"required": {

"tnorm": {"type": str, "choices": {"tnorm1", "tnorm2", "tnorm3"}},

"custom\_obj": {"type": CustomClass}

},

"optional": {"verbose": {"type": bool, "default": False}},

},

"VQRS": {

"class": VQRS,

"required": {

"alpha\_lower": {"type": float},

"beta\_lower": {"type": float}

},

"optional": {"gamma": {"type": float, "default": 0.5}},

},

},

...

}

### 3. **Generic Builder/Validator for Any Module**

Put this in validators.py:

python

CopyEdit

def build\_module(selection: str, config: dict, \*\*kwargs):

if selection not in config:

raise ValueError(f"Unknown selection '{selection}'")

conf = config[selection]

cls = conf["class"]

validated = {}

for key, spec in conf.get("required", {}).items():

if key not in kwargs:

raise ValueError(f"Missing required parameter '{key}' for '{selection}'")

val = kwargs[key]

if "choices" in spec and val not in spec["choices"]:

raise ValueError(f"Invalid value for '{key}': {val}")

if not isinstance(val, spec["type"]):

raise TypeError(f"Expected type {spec['type'].\_\_name\_\_} for '{key}'")

validated[key] = val

for key, spec in conf.get("optional", {}).items():

val = kwargs.get(key, spec["default"])

if not isinstance(val, spec["type"]):

try:

val = spec["type"](val)

except Exception:

raise TypeError(f"Invalid optional parameter '{key}'")

validated[key] = val

return cls(\*\*validated)

### 4. **Build FRSMOTE with This**

python

CopyEdit

from model\_configs import model\_config

from validators import build\_module

class FRSMOTE:

def \_\_init\_\_(self, fr\_model: str, hgf: str, \*\*kwargs):

self.fr\_model = build\_module(fr\_model, model\_config["fr\_model"], \*\*kwargs)

self.hgf = build\_module(hgf, model\_config["hgf"], \*\*kwargs)

def run(self):

print("Running with:", self.fr\_model, self.hgf)

### ✅ Example Usage

python

CopyEdit

from your\_classes import CustomClass

frsmote = FRSMOTE(

fr\_model="ITFRS",

hgf="HGF1",

tnorm="tnorm2",

custom\_obj=CustomClass("some\_setting"),

learning\_rate=0.01

)

frsmote.run()

## ✅ Advantages of This Architecture

| **Feature** | **Benefit** |
| --- | --- |
| Modular | Easy to extend with new models |
| Configurable | Declarative logic, no hardcoding |
| Reusable | Works across many classes |
| Testable | Unit test each module individually |
| Scalable | Nest models within models easily |

## 📦 Optional Enhancements

* ✅ Load config from JSON/YAML for external flexibility.
* ✅ Use pydantic or attrs for stricter validation.
* ✅ Add logging/trace for parameter pipeline.

Would you like a runnable template of this architecture?

# PYDANTIC

Use pydantic or attrs for stricter validation.

###### ChatGPT said:

Excellent — using pydantic or attrs will make your configuration and parameter validation **more robust**, **typed**, and **self-documenting**.

Below, I’ll walk you through how to use **pydantic** (recommended for most use cases) in this modular model selection framework.

## ✅ Why pydantic

* ✅ Auto-validates types and constraints
* ✅ Provides defaults
* ✅ Raises informative errors
* ✅ Can nest models (great for submodels like ITFRS inside FRSMOTE)

## 🧱 Step-by-Step with pydantic

### 1. Install Pydantic

bash

CopyEdit

pip install pydantic

### 2. Define Component Models

Use pydantic.BaseModel for each configurable component.

python

CopyEdit

from pydantic import BaseModel, Field

from typing import Literal, Optional

class CustomClass:

def \_\_init\_\_(self, name: str):

self.name = name

class ITFRSConfig(BaseModel):

tnorm: Literal["tnorm1", "tnorm2", "tnorm3"]

custom\_obj: CustomClass

verbose: bool = False

class VQRSConfig(BaseModel):

alpha\_lower: float

beta\_lower: float

gamma: float = 0.5

### 3. Create a Union Model for fr\_model Selection

You can use a simple function to map model names to config types:

python

CopyEdit

FR\_MODEL\_MAP = {

"ITFRS": ITFRSConfig,

"VQRS": VQRSConfig,

}

### 4. HGF Config Models (Optional)

python

CopyEdit

class HGF1Config(BaseModel):

learning\_rate: float

momentum: float = 0.9

class HGF2Config(BaseModel):

decay: float = 0.01

HGF\_MODEL\_MAP = {

"HGF1": HGF1Config,

"HGF2": HGF2Config,

}

### 5. Generic Factory Function

python

CopyEdit

def build\_model(name: str, config\_map: dict, \*\*kwargs):

if name not in config\_map:

raise ValueError(f"Unknown model type: {name}")

model\_cls = config\_map[name]

return model\_cls(\*\*kwargs)

### 6. Main Class (e.g. FRSMOTE) with Nested Validation

python

CopyEdit

class FRSMOTE:

def \_\_init\_\_(self, fr\_model: str, hgf: str, \*\*kwargs):

self.fr\_model\_name = fr\_model

self.hgf\_name = hgf

self.fr\_model\_config = build\_model(fr\_model, FR\_MODEL\_MAP, \*\*kwargs)

self.hgf\_config = build\_model(hgf, HGF\_MODEL\_MAP, \*\*kwargs)

def run(self):

print(f"Running FRSMOTE with:")

print(f" FR model ({self.fr\_model\_name}): {self.fr\_model\_config}")

print(f" HGF model ({self.hgf\_name}): {self.hgf\_config}")

### ✅ Example Usage

python

CopyEdit

custom\_obj = CustomClass("example")

frsmote = FRSMOTE(

fr\_model="ITFRS",

hgf="HGF1",

tnorm="tnorm2",

custom\_obj=custom\_obj,

learning\_rate=0.01

)

frsmote.run()

### 🧪 Output

plaintext

CopyEdit

Running FRSMOTE with:

FR model (ITFRS): tnorm='tnorm2' custom\_obj=<CustomClass object> verbose=False

HGF model (HGF1): learning\_rate=0.01 momentum=0.9

## ✅ Advantages

| **Feature** | **Benefit** |
| --- | --- |
| ✅ Typed validation | Ensures correctness early |
| ✅ Defaults | Automatically handled |
| ✅ Extensibility | Easily add new models or parameters |
| ✅ Nesting | Supports deeply nested configurations |
| ✅ IDE support | Autocomplete and static typing help |

## Optional: Use attrs Instead?

If you prefer lightweight, non-validating classes (faster), attrs is an alternative. Let me know and I can show the attrs version too.

Would you like a complete file template of this architecture (with tests and runnable CLI)?

# difference between cls and self in python

**ChatGPT said:**

In Python, self and cls are **conventionally used names** that refer to different contexts in object-oriented programming.

**✅ self — Refers to the instance of a class**

* Used in **instance methods**
* Gives access to the object's attributes and methods

python

CopyEdit

class Car:

def \_\_init\_\_(self, color):

self.color = color # self refers to the instance

def drive(self):

print(f"The {self.color} car is driving.")

python

CopyEdit

my\_car = Car("red")

my\_car.drive() # self points to `my\_car`

**✅ cls — Refers to the class itself**

* Used in **class methods** (defined with @classmethod)
* Can be used to **access class variables** or to create new instances

python

CopyEdit

class Car:

wheels = 4

@classmethod

def get\_wheels(cls):

return cls.wheels # cls refers to Car class

python

CopyEdit

Car.get\_wheels() # returns 4

**✅ When to Use Which?**

| **Keyword** | **Used In** | **Refers To** | **Typical Use** |
| --- | --- | --- | --- |
| self | instance methods | object instance | Access or modify instance data |
| cls | class methods | class itself | Access class-level attributes, factories |

**🔧 Example with Both**

python

CopyEdit

class Car:

brand = "Toyota"

def \_\_init\_\_(self, color):

self.color = color # instance variable

@classmethod

def from\_color(cls, color):

return cls(color) # create instance using class reference