

- Read
  - Write
  - Glob
  - Grep
- 

# DSPy Signature Designer

## Goal

Design clear, type-safe signatures that define what your DSPy modules should do.

## When to Use

- Defining new DSPy modules
- Need structured/validated outputs
- Complex input/output relationships
- Multi-field responses

## Inputs

Input	Type	Description
task_description	str	What the module should do
input_fields	list	Required inputs
output_fields	list	Expected outputs
type_constraints	dict	Type hints for fields

## Outputs

Output	Type	Description
signature	dspy.Signature	Type-safe signature class

## Workflow

### Inline Signatures (Simple)

```
import dspy

# Basic
qa = dspy.Predict("question -> answer")

# With types
classify = dspy.Predict("sentence -> sentiment: bool")

# Multiple fields
rag = dspy.ChainOfThought("context: list[str], question: str -> answer: str")
```

### Class-based Signatures (Complex)

```
from typing import Literal, Optional
import dspy

class EmotionClassifier(dspy.Signature):
    """Classify the emotion expressed in the text."""

    text: str = dspy.InputField(desc="The text to analyze")
    emotion: Literal['joy', 'sadness', 'anger', 'fear', 'surprise'] = dspy.Output-
    Field()
    confidence: float = dspy.OutputField(desc="Confidence score 0-1")
```

## Type Hints Reference

```
from typing import Literal, Optional, List
from pydantic import BaseModel

# Basic types
field: str = dspy.InputField()
field: int = dspy.OutputField()
field: float = dspy.OutputField()
field: bool = dspy.OutputField()

# Collections
field: list[str] = dspy.InputField()
field: List[int] = dspy.OutputField()

# Optional
field: Optional[str] = dspy.OutputField()

# Constrained
field: Literal['a', 'b', 'c'] = dspy.OutputField()

# Pydantic models
class Person(BaseModel):
    name: str
    age: int

field: Person = dspy.OutputField()
```

## Production Examples

---

### Summarization

```
class Summarize(dspy.Signature):
    """Summarize the document into key points."""

    document: str = dspy.InputField(desc="Full document text")
    max_points: int = dspy.InputField(desc="Maximum bullet points", default=5)

    summary: list[str] = dspy.OutputField(desc="Key points as bullet list")
    word_count: int = dspy.OutputField(desc="Total words in summary")
```

### Entity Extraction

```
from pydantic import BaseModel
from typing import List

class Entity(BaseModel):
    text: str
    type: str
    start: int
    end: int

class ExtractEntities(dspy.Signature):
    """Extract named entities from text."""

    text: str = dspy.InputField()
    entity_types: list[str] = dspy.InputField(
        desc="Types to extract: PERSON, ORG, LOC, DATE",
        default=["PERSON", "ORG", "LOC"]
    )

    entities: List[Entity] = dspy.OutputField()
```

### Multi-Label Classification

```
class MultiLabelClassify(dspy.Signature):
    """Classify text into multiple categories."""

    text: str = dspy.InputField()

    categories: list[str] = dspy.OutputField(
        desc="Applicable categories from: tech, business, sports, entertainment"
    )
    primary_category: str = dspy.OutputField(desc="Most relevant category")
    reasoning: str = dspy.OutputField(desc="Explanation for classification")
```

## RAG with Confidence

```
class GroundedAnswer(dspy.Signature):
    """Answer questions using retrieved context with confidence."""

    context: list[str] = dspy.InputField(desc="Retrieved passages")
    question: str = dspy.InputField()

    answer: str = dspy.OutputField(desc="Factual answer from context")
    confidence: Literal['high', 'medium', 'low'] = dspy.OutputField(
        desc="Confidence based on context support"
    )
    source_passage: int = dspy.OutputField(
        desc="Index of most relevant passage (0-based)"
    )
```

## Complete Module with Signature

```

import dspy
from typing import Literal, Optional
import logging

logger = logging.getLogger(__name__)

class AnalyzeSentiment(dspy.Signature):
    """Analyze sentiment with detailed breakdown."""

    text: str = dspy.InputField(desc="Text to analyze")

    sentiment: Literal['positive', 'negative', 'neutral', 'mixed'] = dspy.Output-
    Field()
    score: float = dspy.OutputField(desc="Sentiment score from -1 to 1")
    aspects: list[str] = dspy.OutputField(desc="Key aspects mentioned")
    reasoning: str = dspy.OutputField(desc="Explanation of sentiment")

class SentimentAnalyzer(dspy.Module):
    def __init__(self):
        self.analyze = dspy.ChainOfThought(AnalyzeSentiment)

    def forward(self, text: str):
        try:
            result = self.analyze(text=text)

            # Validate score range
            if hasattr(result, 'score'):
                result.score = max(-1, min(1, float(result.score)))

        return result

    except Exception as e:
        logger.error(f"Analysis failed: {e}")
        return dspy.Prediction(
            sentiment='neutral',
            score=0.0,
            aspects=[],
            reasoning="Analysis failed"
        )

# Usage
analyzer = SentimentAnalyzer()
result = analyzer(text="The product quality is great but shipping was slow.")
print(f"Sentiment: {result.sentiment} ({result.score})")
print(f"Aspects: {result.aspects}")

```

## Best Practices

1. **Descriptive docstrings** - The class docstring becomes the task instruction
2. **Field descriptions** - Guide the model with `desc` parameter
3. **Constrain outputs** - Use `Literal` for categorical outputs
4. **Default values** - Provide sensible defaults for optional inputs
5. **Validate types** - Pydantic models ensure structured output

## Limitations

---

- Complex nested types may require Pydantic
- Some LLMs struggle with strict type constraints
- JSONAdapter works better for structured outputs
- Field descriptions add to prompt length