

✓ DSPy-AI Prompting Cheat Sheet (Beginner → Advanced)

(All examples in Python using DSPy v2+)

DSPy is NOT "prompt engineering."

It is **programming** LLM pipelines using **modules** and **signatures** that guarantee correctness and reproducibility.

▨ 1. BASIC CONCEPTS

✓ 1.1 Importing DSPy

```
import dspy
```

✓ 1.2 Set an LLM

```
dspy.settings.configure(lm=dspy.LM("gpt-4o-mini"))
```

or local LLM:

```
dspy.settings.configure(lm=dspy.LM("ollama:phi3"))
```

✓ 1.3 What is a *Signature*?

A signature defines **input fields** and **output fields**.

```
class Summarize(dspy.Signature):
    """Summarize text into 1 sentence."""
    text: str
    summary: str
```

✓ 1.4 What is a *Module*?

A module uses the signature:

```
class Summarizer(dspy.Module):
    def forward(self, text):
        return dspy.Predict(Summarize)(text=text)
```

✓ 1.5 Calling the model

```
summarizer = Summarizer()
summarizer("Machine learning is cool.")
```

2. PROMPTING WITH SIGNATURES

DSPy DOES NOT use prompt strings.
You describe **what** you want, not **how** to say it.

✓ 2.1 Classification Signature

```
class Sentiment(dspy.Signature):
    """Return 'positive', 'negative', or 'neutral'."""
    text: str
    label: str
```

Usage:

```
predictor = dspy.Predict(Sentiment)
predictor(text="I love DSPy!")
```

✓ 2.2 Multi-output Signature

```
class QA(dspy.Signature):
    question: str
    answer: str
    explanation: str
```

Usage:

```
qa = dspy.Predict(QA)
qa(question="Why is the sky blue?")
```

3. GUIDED PROMPTING (CONSTRAINTS)

DSPy allows **hard constraints** using descriptions and types.

✓ 3.1 Forced format:

```
class CodeGenerator(dspy.Signature):
    """Generate only Python code, no explanation."""
    specification: str
    python_code: dspy.Code
```

✓ 3.2 Forced enumeration:

```
class Category(dspy.Signature):
    """Return one category only."""
    text: str
    label: dspy.Enum("sports", "politics", "tech")
```

✓ 3.3 Forced list type:

```
class Keywords(dspy.Signature):
    text: str
    keywords: list[str]
```

4. OPTIMIZATION (DSPy's Secret Power)

DSPy can optimize prompts automatically using training examples.

Example Training Data

```
train = [  
    dict(text="I love pizza", label="positive"),  
    dict(text="I hate bugs", label="negative"),  
]
```

Compile a Program

```
program = dspy.Predict(Sentiment)
```

Optimizer (bootstrap)

```
optimizer = dspy.BootstrapFewShot(program=program, trainset=train)  
program = optimizer.compile()
```

DSPy now rewrites the prompt automatically.

5. RETRIEVAL-AUGMENTED GENERATION (RAG)

5.1 Pass context into signatures

```
class RAGQA(dspy.Signature):  
    question: str  
    context: str  
    answer: str
```

5.2 Build a module using a retriever

```
class RAG(dspy.Module):
    def __init__(self, retriever):
        super().__init__()
        self.retriever = retriever
        self.qa = dspy.Predict(RAGQA)

    def forward(self, question):
        docs = self.retriever(question)
        context = "\n".join(docs)
        return self.qa(question=question, context=context)
```

Usage:

```
rag = RAG(retriever=my_vectoradb.search)
rag("What is photosynthesis?")
```



6. TOOL USE (FUNCTION CALLING)

DSPy allows structured tool calling.

6.1 Define a tool signature

```
class Calculator(dspy.Tool):
    def add(self, a: int, b: int) -> int:
        return a + b
```

6.2 Use tool in a signature

```
class MathQuestion(dspy.Signature):  
    question: str  
    answer: int
```

6.3 Create module with tool use:

```
class MathAgent(dspy.Module):  
    def __init__(self):  
        super().__init__()  
        self.solve = dspy.Predict(MathQuestion)  
  
    def forward(self, question):  
        return self.solve(question=question)
```

7. ADVANCED AGENTS (Multi-step Reasoning)

7.1 Chain multiple modules

```

class Step1(dspy.Module):
    def forward(self, question):
        return dict(keywords=question.split())

class Step2(dspy.Module):
    def forward(self, keywords):
        return dict(summary=", ".join(keywords))

class Pipeline(dspy.Module):
    def __init__(self):
        self.s1 = Step1()
        self.s2 = Step2()

    def forward(self, question):
        k = self.s1(question)
        return self.s2(**k)

```

Usage:

```

pipe = Pipeline()
pipe("machine learning is awesome")

```

8. FULL REAL ADVANCED AGENT (Planning + Tools + RAG)

This is the type of agent used in DSPy research papers.

Step 1: Signatures

```
class Plan(dspy.Signature):
    goal: str
    steps: list[str]

class Execute(dspy.Signature):
    step: str
    result: str
```

Step 2: Agent

```
class PlannerExecutor(dspy.Module):
    def __init__(self):
        super().__init__()
        self.plan = dspy.Predict(Plan)
        self.exec = dspy.Predict(Execute)

    def forward(self, goal):
        plan = self.plan(goal=goal)
        results = []

        for step in plan.steps:
            r = self.exec(step=step)
            results.append(r.result)

        return {"final_answer": "\n".join(results)}
```

Usage:

```
agent = PlannerExecutor()
agent("Explain how neural networks learn.")
```
