

# Tutorial : Maîtriser le Prompt Engineering Toolkit (PET)

## Introduction

Ce tutorial vous guide étape par étape dans l'utilisation du Prompt Engineering Toolkit (PET). Nous progresserons du niveau débutant au niveau expert avec des exemples concrets et pratiques.

## Prérequis

```
python
```

```
# Assurez-vous d'avoir le module PromptToolkit.py dans votre répertoire
```

```
from PromptToolkit import *
```

## ● NIVEAU 1 : LES BASES - Créer son premier prompt

### 1.1 Construction manuelle simple

**Objectif :** Créer un prompt basique pour résumer un article

```
python
```

```
def niveau_1_prompt_basique():
```

```
    """Exemple le plus simple : création directe d'un prompt"""
```

```
    # Création manuelle d'un prompt
```

```
    mon_prompt = Prompt(
```

```
        instruction="Résumez l'article suivant en 3 points clés",
```

```
        context="Article de presse économique sur l'intelligence artificielle",
```

```
        constraints=["Maximum 100 mots", "Utiliser des puces", "Langage accessible"]
```

```
)
```

```
    # Ajout d'un exemple
```

```
    mon_prompt.add_example("Article sur crypto -> Résumé: • Prix volatil • Adoption croissante • Régulation incertain")
```

```
    print("=== PROMPT BASIQUE ===")
```

```
    print(mon_prompt.format())
```

```
    print(f"Valide: {mon_prompt.validate()}")
```

```
    return mon_prompt
```

💡 **Points clés à retenir :**

- La classe `Prompt` encapsule tous les éléments
- `add_example()` et `add_constraint()` enrichissent le prompt
- `validate()` vérifie la qualité
- `format()` génère le prompt final prêt à utiliser

## 1.2 Utilisation d'un template simple

**Objectif :** Utiliser un template prédéfini

```
python

def niveau_1_avec_template():
    """Utilisation d'un template classification simple"""

    # Template prêt à l'emploi
    classifieur_sentiment = classification_template(
        categories=["Positif", "Neutre", "Négatif"],
        description="Classification de commentaires clients sur un produit",
        exemples=[
            ("Ce produit est fantastique, je le recommande!", "Positif"),
            ("Livraison correcte, rien d'exceptionnel", "Neutre"),
            ("Service client décevant, je ne rachèterai pas", "Négatif")
        ]
    )

    print("=== TEMPLATE CLASSIFICATION ===")
    print(classifieur_sentiment.format())

    return classifieur_sentiment
```

### 💡 Ce que vous apprenez :

- Les templates accélèrent le développement
- Ils intègrent les bonnes pratiques automatiquement
- Les exemples sont formatés correctement

---

## ● NIVEAU 2 : TRANSFORMATIONS - Améliorer ses prompts

### 2.1 Transformation simple

**Objectif :** Modifier un prompt existant avec une transformation

python

```
def niveau_2_transformation_simple():  
    """Application d'une transformation simple"""  
  
    # Prompt de base  
    prompt_base = Prompt(  
        instruction="Expliquez ce concept technique à un débutant",  
        context="Concept: Les microservices en architecture logicielle"  
    )  
  
    # Application d'une transformation  
    prompt_formel = make_formal(prompt_base)  
  
    print("=== AVANT TRANSFORMATION ===")  
    print(prompt_base.format())  
  
    print("\n=== APRÈS TRANSFORMATION (formal) ===")  
    print(prompt_formel.format())  
  
    # Comparaison  
    print(f"\nPrompt original valide: {prompt_base.validate()}")  
    print(f"Prompt transformé valide: {prompt_formel.validate()}")  
  
    return prompt_base, prompt_formel
```

## 2.2 Pipeline de transformations

**Objectif :** Combiner plusieurs transformations en séquence

python

```
def niveau_2_pipeline():  
    """Utilisation d'un pipeline de transformations multiples"""  
  
    # Prompt initial simple  
    prompt_analyse = Prompt(  
        instruction="Analysez les risques de ce projet",  
        context="Projet: Migration de 50 applications vers le cloud en 6 mois"  
    )  
  
    # Création d'un pipeline de transformations  
    pipeline_analyse = pipeline(  
        make_formal,          # 1. Rendre formel  
        add_confidence_scoring, # 2. Ajouter scoring confiance  
        lambda p: add_explanation_requirement( # 3. Demander justifications  
            p, "Justifiez chaque risque identifié:"  
        )  
    )  
  
    # Application du pipeline  
    prompt_enrichi = pipeline_analyse(prompt_analyse)  
  
    print("=== PIPELINE DE TRANSFORMATIONS ===")  
    print(prompt_enrichi.format())  
  
    return prompt_enrichi
```

### 💡 Points d'apprentissage :

- `pipeline()` combine plusieurs transformations
- L'ordre des transformations compte
- Les lambda permettent de personnaliser les transformations

## 🟠 NIVEAU 3 : TEMPLATES AVANCÉS - Cas d'usage spécialisés

### 3.1 Template Few-Shot Learning

**Objectif :** Créer un système d'apprentissage par l'exemple

python

```
def niveau_3_few_shot():  
    """Template few-shot pour reconnaissance de patterns"""  
  
    # Exemples pour apprendre un pattern de bug reports  
    exemples_bugs = [  
        (  
            "L'app crash quand je clique sur 'Envoyer' après avoir tapé un message long",  
            "BUG: UI/UX | IMPACT: Bloquant | REPRO: Message >200 chars + bouton Envoyer | PRIORITÉ: Haute"  
        ),  
        (  
            "Les images ne se chargent pas bien sur mobile, ça prend 30 secondes",  
            "BUG: Performance | IMPACT: Gênant | REPRO: Mobile + chargement images | PRIORITÉ: Moyenne"  
        ),  
        (  
            "Impossible de se connecter depuis hier soir, erreur 500",  
            "BUG: Backend | IMPACT: Critique | REPRO: Connexion utilisateur | PRIORITÉ: Critique"  
        )  
    ]  
  
    # Template few-shot  
    bug_classifier = few_shot_learning_template(  
        task="classification et structuration de bug reports",  
        exemples=exemples_bugs,  
        pattern_description="Format: TYPE | IMPACT | REPRODUCTION | PRIORITÉ"  
    )  
  
    # Enrichissement avec pipeline  
    bug_enhanced = pipeline(  
        add_confidence_scoring,  
        lambda p: add_explanation_requirement(p, "Expliquez votre classification:")  
    )(bug_classifier)  
  
    print("=== FEW-SHOT BUG CLASSIFICATION ===")  
    print(bug_enhanced.format())  
  
    return bug_enhanced
```

## 3.2 Template Chain-of-Thought

**Objectif :** Prompt pour raisonnement étape par étape

python

```
def niveau_3_chain_of_thought():  
    """Template pour résolution de problème complexe"""  
  
    # Problème business complexe  
    problem_solver = chain_of_thought_template(  
        problem_type="optimisation business",  
        context="""  
        Situation: Une startup SaaS (ARR: 2M€) constate:  
        - Churn rate: 15% mensuel (industrie: 5-7%)  
        - Customer Acquisition Cost: 800€ (LTV: 2400€)  
        - Support tickets: +40% en 3 mois  
        - Équipe dev: surchargée, vitesse -30%  
  
        Question: Quelle stratégie adopter en priorité?  
        """)  
    )  
  
    # Pipeline d'enrichissement pour analyse business  
    business_pipeline = pipeline(  
        lambda p: add_alternative_perspectives(p, 2), # Perspectives alternatives  
        lambda p: add_source_requirements(p, [ # Sources requises  
            "métriques SaaS", "benchmarks secteur", "analyses churn"  
        ]),  
        add_confidence_scoring  
    )  
  
    problem_enhanced = business_pipeline(problem_solver)  
  
    print("=== CHAIN-OF-THOUGHT BUSINESS ===")  
    print(problem_enhanced.format())  
  
    return problem_enhanced
```

### 💡 Concepts avancés :

- Few-shot learning : l'IA apprend des patterns à partir d'exemples
- Chain-of-thought : décomposition du raisonnement en étapes
- Perspectives alternatives enrichissent l'analyse

## ● NIVEAU 4 : ARCHITECTURES COMPLEXES - Systèmes multi-prompts

### 4.1 Génération de variations pour A/B Testing

**Objectif :** Créer plusieurs versions d'un prompt pour tester leur efficacité

python

```
def niveau_4_variations_ab():  
    """Génération systématique de variations pour tests A/B"""  
  
    # Prompt de base pour customer success  
    base_customer_success = Prompt(  
        instruction="Rédigez un email de réactivation pour un client inactif depuis 60 jours",  
        context="""  
        Client: PME utilisant notre CRM depuis 18 mois  
        Dernière connexion: il y a 60 jours  
        Historique: Utilisateur actif puis arrêt brutal  
        Objectif: Identifier les blocages et proposer solutions  
        """,  
        constraints=[  
            "Ton empathique mais professionnel",  
            "Maximum 200 mots",  
            "Call-to-action clair"  
        ]  
    )  
  
    # Stratégies de variation différentes  
    strategies_ab = [  
        # Variation A: Approche directe  
        lambda p: pipeline(  
            make_formal,  
            lambda p: add_explanation_requirement(p, "Utilisez une approche directe et factuelle")  
        )(p),  
  
        # Variation B: Approche empathique  
        lambda p: pipeline(  
            lambda p: add_explanation_requirement(p, "Montrez de l'empathie et de la compréhension"),  
            lambda p: add_alternative_perspectives(p, 1)  
        )(p),  
  
        # Variation C: Approche value-driven  
        lambda p: pipeline(  
            lambda p: add_explanation_requirement(p, "Mettez l'accent sur la valeur perdue"),  
            add_confidence_scoring  
        )(p),  
  
        # Variation D: Approche socratique  
        lambda p: pipeline(  
            make_socratic,  
            lambda p: add_explanation_requirement(p, "Posez des questions pour comprendre")  
        )(p)  
    ]  
  
    # Génération des variations
```

```
variations = generate_variations(base_customer_success, strategies_ab)
```

```
print("=== VARIATIONS POUR A/B TESTING ===")
for i, variation in enumerate(variations, 1):
    print(f"\n--- VARIATION {chr(64+i)} ---")
    print(f"Stratégie: {'Directe', 'Empathique', 'Value-driven', 'Socratique'}[{i-1}]")
    print(variation.format()[300] + "...")
    print(f"Valide: {'✓' if variation.validate() else 'X'}")

return variations
```

## 4.2 Système multi-templates avec orchestration

**Objectif :** Combiner plusieurs templates pour un workflow complexe



python

```
def niveau_4_workflow_complexe():
    """Workflow d'analyse complète avec multiple templates"""

    print("=== WORKFLOW ANALYSE PRODUIT COMPLEXE ===")

    # ÉTAPE 1: Analyse SWOT
    swot_analysis = structured_analysis_template(
        subject="Lancement d'une plateforme de formation IA pour entreprises",
        framework="Analyse SWOT",
        dimensions=["Forces", "Faiblesses", "Opportunités", "Menaces"]
    )

    # ÉTAPE 2: Scénarios de marché
    market_scenarios = scenario_planning_template(
        situation="Pénétration marché formation IA entreprise",
        variables=["adoption IA", "budgets formation", "concurrence", "réglementation"],
        time_horizon="18 mois"
    )

    # ÉTAPE 3: Comparaison concurrentielle
    competitor_analysis = comparison_template(
        items=["Coursera Business", "LinkedIn Learning", "Udacity Enterprise"],
        aspects=["prix", "contenu IA", "certification", "intégration LMS", "support client"]
    )

    # ÉTAPE 4: Plan d'action
    action_plan = role_play_template(
        role="directeur produit expérimenté",
        scenario="Synthèse des analyses précédentes pour définir la roadmap produit",
        objectives=[
            "Prioriser les fonctionnalités critiques",
            "Identifier les risques majeurs",
            "Définir les métriques de succès",
            "Planifier les 6 premiers mois"
        ]
    )

    # Pipeline d'enrichissement commun
    enrichment_pipeline = pipeline(
        make_formal,
        add_confidence_scoring,
        lambda p: add_source_requirements(p, ["études marché", "benchmarks", "données clients"])
    )

    # Application du pipeline à tous les templates
    workflow_prompts = {
        "swot": enrichment_pipeline(swot_analysis),
        "scenarios": enrichment_pipeline(market_scenarios),
        "competitor": enrichment_pipeline(competitor_analysis),
        "action": enrichment_pipeline(action_plan)
    }
```

```
"scenarios": enrichment_pipeline(market_scenarios),  
"competition": enrichment_pipeline(competitor_analysis),  
"action_plan": enrichment_pipeline(action_plan)  
}
```

*# Affichage du workflow*

```
for step_name, prompt in workflow_prompts.items():  
    print(f"\n{'='*20} ÉTAPE: {step_name.upper()} {'='*20}")  
    print(prompt.format():[:400] + "... \n")  
    print(f"Validation: {'✓' if prompt.validate() else 'X'}")  
  
return workflow_prompts
```

### 💡 Architecture avancée :

- Orchestration de multiples templates
- Pipeline réutilisable sur plusieurs prompts
- Workflow structuré pour analyses complexes

---

## 🟪 NIVEAU 5 : EXPERT - Gestion et optimisation

### 5.1 Système de sauvegarde et réutilisation

**Objectif :** Créer une bibliothèque de prompts réutilisables

python

```
import json
```

```
from datetime import datetime
```

```
def niveau_5_bibliotheque_prompts():
```

```
    """Système avancé de gestion de bibliothèque de prompts"""
```

```
    # Classe pour gérer une bibliothèque
```

```
    class PromptLibrary:
```

```
        def __init__(self):
```

```
            self.prompts = {}
```

```
            self.metadata = {
```

```
                "created": datetime.now().isoformat(),
```

```
                "version": "1.0",
```

```
                "total_prompts": 0
```

```
            }
```

```
        def add_prompt(self, name, prompt, tags=None, description=""):
```

```
            """Ajoute un prompt à la bibliothèque"""
```

```
            self.prompts[name] = {
```

```
                "prompt_data": prompt.to_dict(),
```

```
                "tags": tags or [],
```

```
                "description": description,
```

```
                "created": datetime.now().isoformat(),
```

```
                "usage_count": 0
```

```
            }
```

```
            self.metadata["total_prompts"] = len(self.prompts)
```

```
        def get_prompt(self, name):
```

```
            """Récupère et instancie un prompt"""
```

```
            if name in self.prompts:
```

```
                self.prompts[name]["usage_count"] += 1
```

```
                return Prompt.from_dict(self.prompts[name]["prompt_data"])
```

```
            return None
```

```
        def search_by_tags(self, tags):
```

```
            """Recherche par tags"""
```

```
            results = []
```

```
            for name, data in self.prompts.items():
```

```
                if any(tag in data["tags"] for tag in tags):
```

```
                    results.append((name, data["description"]))
```

```
            return results
```

```
        def export_json(self, filename):
```

```
            """Export en JSON"""
```

```
            full_data = {
```

```
                "metadata": self.metadata,
```

```
                "prompts": self.prompts
```

```
            }
```

```
}  
with open(filename, 'w', encoding='utf-8') as f:  
    json.dump(full_data, f, indent=2, ensure_ascii=False)
```

*# Création de la bibliothèque*

```
lib = PromptLibrary()
```

*# Ajout de prompts spécialisés*

*# 1. Prompt d'analyse de code*

```
code_analyzer = error_analysis_template(  
    domain="Python/Django",  
    error_types=["sécurité", "performance", "maintenabilité", "bonnes pratiques"]  
)  
lib.add_prompt(  
    "code_analyzer_python",  
    code_analyzer,  
    tags=["développement", "code", "python", "analyse"],  
    description="Analyseur de code Python/Django complet"  
)
```

*# 2. Générateur de documentation*

```
doc_generator = text_generation_template(  
    style="technique documentaire",  
    requirements=["Structure claire", "Exemples concrets", "API reference"],  
    length_constraint="500-1000 mots"  
)  
lib.add_prompt(  
    "doc_generator",  
    doc_generator,  
    tags=["documentation", "technique", "API"],  
    description="Générateur de documentation technique"  
)
```

*# 3. Analyseur de sentiment client*

```
sentiment_classifier = few_shot_learning_template(  
    task="analyse de sentiment client avancée",  
    examples=[  
        ("Service rapide mais produit décevant", "MITIGE: Service(+) Produit(-)"),  
        ("Excellent support, résolution immédiate!", "POSITIF: Support(++) Résolution(++)" ),  
        ("Facturation confuse, j'abandonne", "NEGATIF: Process(-) Intention_exit")  
    ],  
    pattern_description="Sentiment global + détail par aspect"  
)  
lib.add_prompt(  
    "sentiment_analyzer_advanced",  
    sentiment_classifier,  
    tags=["client", "sentiment", "feedback", "analyse"],  
    description="Analyse de sentiment client avec détail par aspects"  
)
```

```

print("=== BIBLIOTHÈQUE DE PROMPTS ===")
print(f"Prompts créés: {lib.metadata['total_prompts']}")

# Démonstration de recherche
results = lib.search_by_tags(["analyse", "code"])
print(f"\nRecherche 'analyse + code': {len(results)} résultats")
for name, desc in results:
    print(f" - {name}: {desc}")

# Réutilisation d'un prompt
retrieved_prompt = lib.get_prompt("code_analyzer_python")
if retrieved_prompt:
    print(f"\nPrompt récupéré et utilisable:")
    print(f"Valid: {retrieved_prompt.validate()}")

# Modification pour nouveau contexte
retrieved_prompt.context = "Code Django avec vulnérabilité potentielle CSRF"
print("Prompt adapté pour nouveau contexte ✓")

# Export de la bibliothèque
lib.export_json("ma_bibliotheque_prompts.json")
print("\nBibliothèque exportée en JSON ✓")

return lib

```

## 5.2 Métriques et optimisation de prompts

**Objectif :** Mesurer et optimiser la performance des prompts

python

```
def niveau_5_optimisation():
    """Système de métriques et d'optimisation de prompts"""

    class PromptOptimizer:
        def __init__(self):
            self.metrics = {}

        def analyze_prompt_quality(self, prompt, name):
            """Analyse la qualité d'un prompt selon plusieurs critères"""
            metrics = {
                "clarity_score": self._calculate_clarity(prompt),
                "complexity_score": self._calculate_complexity(prompt),
                "completeness_score": self._calculate_completeness(prompt),
                "reusability_score": self._calculate_reusability(prompt)
            }

            # Score global
            metrics["overall_score"] = sum(metrics.values()) / len(metrics)
            self.metrics[name] = metrics
            return metrics

        def _calculate_clarity(self, prompt):
            """Score de clarté (0-10)"""
            score = 5.0

            # Instruction claire et spécifique
            if len(prompt.instruction) > 20:
                score += 1

            if any(verb in prompt.instruction.lower() for verb in
                ["analyze", "generate", "classify", "compare", "explain"]):
                score += 1

            # Contraintes définies
            if prompt.constraints:
                score += 1

            # Format de sortie spécifié
            if prompt.output_format:
                score += 1

            return min(10, score)

        def _calculate_complexity(self, prompt):
            """Score de complexité appropriée (0-10)"""
            score = 5.0

            # Équilibre des éléments
```

```

total_elements = len(prompt.constraints) + len(prompt.examples)
if 2 <= total_elements <= 8:
    score += 2
elif total_elements > 8:
    score -= 1 # Trop complexe

# Contexte approprié
if prompt.context and len(prompt.context) < 500:
    score += 1

return min(10, score)

def _calculate_completeness(self, prompt):
    """Score de complétude (0-10)"""
    score = 0

    # Éléments essentiels
    if prompt.instruction: score += 3
    if prompt.context: score += 2
    if prompt.constraints: score += 2
    if prompt.examples: score += 2
    if prompt.output_format: score += 1

    return min(10, score)

def _calculate_reusability(self, prompt):
    """Score de réutilisabilité (0-10)"""
    score = 5.0

    # Généricité vs spécificité
    if prompt.context and "{" in prompt.context: # Variables
        score += 2
    if len(prompt.examples) >= 2:
        score += 1
    if prompt.metadata:
        score += 1

    return min(10, score)

def suggest_improvements(self, prompt, metrics):
    """Suggestions d'amélioration basées sur les métriques"""
    suggestions = []

    if metrics["clarity_score"] < 7:
        suggestions.append("Clarifier l'instruction avec des verbes d'action spécifiques")
        suggestions.append("Ajouter un format de sortie détaillé")

    if metrics["completeness_score"] < 7:
        suggestions.append("Ajouter des exemples concrets")
        suggestions.append("Définir des contraintes plus précises")

```

```

suggestions.append("Ajouter des métadonnées plus précises")

if metrics["reusability_score"] < 6:
    suggestions.append("Généraliser le contexte avec des variables")
    suggestions.append("Ajouter des métadonnées pour le catalogage")

return suggestions

# Test du système d'optimisation
optimizer = PromptOptimizer()

# Prompt à optimiser
prompt_test = Prompt(
    instruction="Faire quelque chose avec ce texte", # Volontairement vague
    context="Un texte quelconque"
)

print("=== ANALYSE DE QUALITÉ ===")
metrics = optimizer.analyze_prompt_quality(prompt_test, "prompt_vague")

for metric, score in metrics.items():
    print(f"{metric}: {score:.1f}/10")

# Suggestions d'amélioration
suggestions = optimizer.suggest_improvements(prompt_test, metrics)
print(f"\n=== SUGGESTIONS D'AMÉLIORATION ===")
for i, suggestion in enumerate(suggestions, 1):
    print(f"{i}. {suggestion}")

# Version améliorée
prompt_ameliore = classification_template(
    categories=["Technique", "Business", "Créatif"],
    description="Classification de contenus selon leur nature",
    examples=[
        ("Documentation API REST", "Technique"),
        ("Plan marketing Q4", "Business"),
        ("Brainstorming noms produit", "Créatif")
    ]
)






print(f"\n=== APRÈS AMÉLIORATION ===")
metrics_ameliore = optimizer.analyze_prompt_quality(prompt_ameliore, "prompt_ameliore")
for metric, score in metrics_ameliore.items():
    print(f"{metric}: {score:.1f}/10")

return optimizer

```



## Progression recommandée :

1.  **Niveau 1** : Maîtrisez la création manuelle et les templates simples
2.  **Niveau 2** : Explorez les transformations et pipelines
3.  **Niveau 3** : Utilisez les templates avancés (few-shot, chain-of-thought)
4.  **Niveau 4** : Orchestrez des workflows complexes
5.  **Niveau 5** : Optimisez et gérez vos bibliothèques de prompts

## Conseils d'expert :

- ✓ **Toujours valider** vos prompts avec `validate()`
- ✓ **Tester en conditions réelles** avant de déployer
- ✓ **Documenter vos templates** réutilisables
- ✓ **Mesurer la performance** avec des métriques
- ✓ **Itérer et améliorer** continuellement

## Patterns à retenir :

- **Pipeline ➤ Transformation unique** : Combinez toujours plusieurs améliorations
- **Template + Pipeline** : Partez d'un template et enrichissez-le
- **Few-shot ➤ Zero-shot** : Les exemples améliorent drastiquement la qualité
- **Validation systématique** : Un prompt non validé est un risque

---

## ALLER PLUS LOIN

Une fois ces concepts maîtrisés, vous pouvez :

- Créer vos propres transformations personnalisées
- Développer des templates métier spécifiques
- Intégrer des APIs pour l'évaluation automatique
- Construire des interfaces utilisateur pour vos prompts
- Implémenter des systèmes de versioning avancés

Le Toolkit PET vous donne toutes les briques pour devenir un expert en Prompt Engineering ! 