

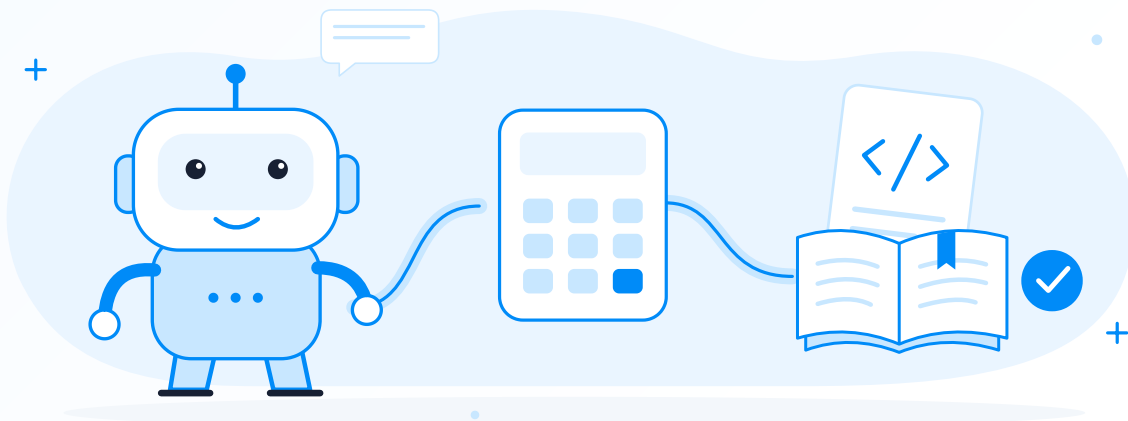
Tutoriel

Agents · Function calling · Python

Créer un agent avec deux outils locaux

Construire un agent minimal qui choisit entre une calculatrice et une recherche dans un catalogue, sans lui donner de droit d'écriture.

Auteur **Youness Boutyour**

**DURÉE**

60 minutes

NIVEAU

Intermédiaire

FORMAT

Tutoriel pratique

Agents · Function calling · Python · publié le 2026-08-01 · mock

Objectifs et livrables

Un assistant de formation reçoit deux demandes très différentes : calculer une moyenne et retrouver un cours dans un catalogue. Nous allons lui donner les moyens d'y répondre en construisant un petit agent Python doté de deux outils, `calculate(expression)` et `find_course(topic)`.

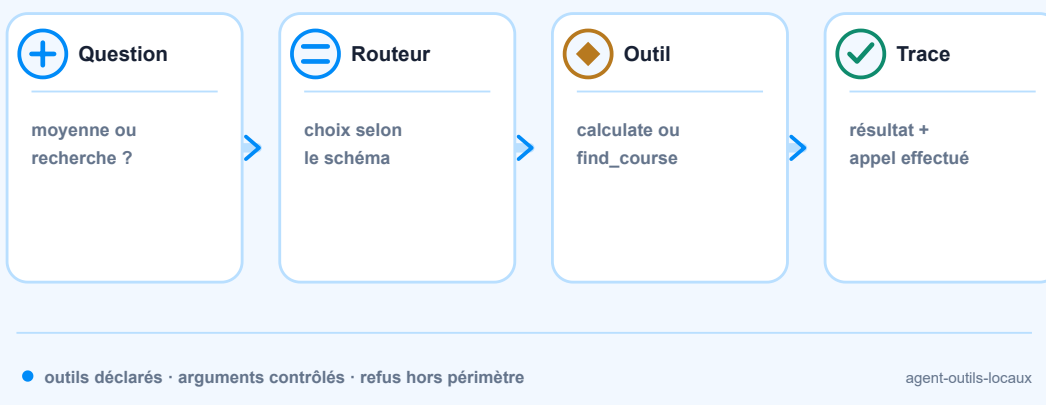
Vous commencerez par écrire les fonctions et définir leurs entrées, puis vous les rendrez accessibles à la boucle de l'agent. Nous suivrons une demande depuis le choix de l'outil jusqu'à la réponse, en passant par les arguments et le résultat retourné. Le mode de démonstration permet d'examiner ce parcours de façon reproductible.

L'intérêt de cette séparation apparaît dès les premiers essais : vous pouvez tester le calcul et la recherche indépendamment de la décision de l'agent. Nous examinerons aussi une demande hors périmètre pour vérifier qu'elle est refusée. Les capacités disponibles restent limitées aux fonctions déclarées, sans accès libre aux commandes, aux URL ou à une base de données.

Le fichier `outputs/trace.jsonl` réunira les demandes et les appels effectués. En le lisant, vous pourrez expliquer quel outil a répondu, avec quelles données, et comment son résultat a servi à construire la réponse finale.

Routage contrôlé vers un outil local

Schéma de l'agent local : une question est classée, envoyée vers un outil autorisé, puis conservée dans une trace.



Le routeur ne donne pas un accès général : il choisit parmi les fonctions déclarées et journalisées.

Lire le flux avant de coder

Le schéma présente une décision bornée : l'agent ne choisit pas une commande arbitraire, il sélectionne une fonction parmi deux outils déclarés. Les arguments sont ensuite transmis au code local, qui reste responsable de la validation et du format de retour.

Le contrôle le plus révélateur est la demande hors périmètre. Elle doit être refusée sans tentative d'accès supplémentaire. En lisant `trace.jsonl`, vous pouvez relier la question, le nom de l'outil, ses arguments et la réponse obtenue ; cette chaîne est plus utile qu'une réponse finale isolée.

1. Préparer l'environnement

Gardez les fichiers de l'archive dans leur arborescence : le lanceur, les outils et les fixtures remplissent des rôles différents. Commencez par le scénario mock pour vérifier que le programme importe les outils et écrit dans `outputs/`. Vous pourrez ensuite modifier une demande à la fois ; si vous changez simultanément les outils et le scénario, la trace sera plus difficile à interpréter.

L'environnement virtuel sert à isoler les dépendances du projet. Une fois activé, la commande `python -m pip` installe les paquets pour ce même interpréteur ; le test d'import vérifie donc le Python qui exécutera l'agent. Gardez le terminal dans le dossier du projet pour que les chemins vers `src/` et `outputs/` restent corrects.

La calculatrice ci-dessous commence par analyser l'expression avec `ast.parse`. La fonction parcourt ensuite uniquement les nombres et les opérations présents dans `BINOPS`. Lire ce parcours permet de comprendre pourquoi une addition est acceptée et pourquoi un appel de fonction Python doit être refusé. Testez les fonctions locales avant de les confier au mécanisme de sélection de l'agent.

Terminal · Bash

```
mkdir -p agent-outils-locaux/{src,outputs}
cd agent-outils-locaux
python3 -m venv .venv && source .venv/bin/activate
python -m pip install pytest
```

Sortie obtenue

Sortie / exemple

Successfully installed pytest

Commande de vérification

Terminal · Bash

```
python -c "import pytest; print('agent stack ok')"
```

Code de l'étape

Créez d'abord les fonctions locales dans `src/agent.py` :

Python

```
import ast
import operator

BINOPS = {ast.Add: operator.add, ast.Sub: operator.sub,
          ast.Mult: operator.mul, ast.Div: operator.truediv}

def calculate(expression):
    def visit(node):
        if isinstance(node, ast.Constant) and isinstance(node.value, (int, float)):
            return node.value
        if isinstance(node, ast.UnaryOp) and isinstance(node.op, (ast.UAdd, ast.USub)):
            value = visit(node.operand)
            return value if isinstance(node.op, ast.UAdd) else -value
        if isinstance(node, ast.BinOp) and type(node.op) in BINOPS:
            return BINOPS[type(node.op)](visit(node.left), visit(node.right))
        raise ValueError("expression interdite")
    return visit(ast.parse(expression, mode="eval").body)

def find_course(topic):
    catalogue = {"rag": ["RAG local", "Recherche sémantique"]}
    return next((items for key, items in catalogue.items() if key in topic.casefold()), [])
```

2. Déclarer les outils autorisés

Testez chaque fonction directement avant de la confier à la boucle d'agent : une expression simple pour le calcul et un terme présent dans le catalogue pour la recherche. Le calculateur parcourt uniquement les opérations AST autorisées ; ce n'est pas un interpréteur Python général. Une division par zéro reste une erreur d'exécution à traiter, même lorsque l'expression respecte cette liste d'opérations.

Le dictionnaire `TOOLS` associe un nom public à une fonction Python précise. C'est cette table qui détermine ce que le programme peut réellement appeler : un nom proposé par un modèle ne devient pas automatiquement une nouvelle capacité. La vérification de présence dans le registre doit donc précéder l'appel.

Les arguments sont transmis avec `**arguments`, ce qui suppose un dictionnaire dont les clés correspondent aux paramètres de la fonction. Distinguez une fonction inconnue d'un argument incorrect : la première doit être rejetée par le registre, tandis que le second nécessite une validation du contrat d'entrée. L'essai avec `delete_file` vérifie que le refus se produit avant toute exécution.

Créez un registre explicite :

Code de l'étape

Python

```
TOOLS = {"calculate": calculate, "find_course": find_course}

def dispatch(name, arguments):
    if name not in TOOLS:
        raise ValueError(f"outil interdit: {name}")
    return TOOLS[name](**arguments)
```

Sortie obtenue

Une fonction inconnue est refusée sans être appelée.

Sortie / exemple

```
ValueError: outil interdit: delete_file
trace: 0 appel d'outil externe
```

Commande de vérification

Terminal · Bash

```
python src/agent.py --mock --tool delete_file; test $? -ne 0
```

3. Exécuter la boucle agent-outil

Suivez les variables dans l'ordre : nom demandé, arguments reçus, fonction sélectionnée, puis résultat retourné. Vérifier que les arguments sont un dictionnaire ne suffit pas à garantir que chaque paramètre a le bon type ou une valeur acceptable. Le répartiteur et la fonction appelée ont donc des responsabilités complémentaires ; une demande refusée ne doit pas être transformée silencieusement en appel d'un autre outil.

Suivez la boucle comme une succession de données : la demande conduit à un nom d'outil et à des arguments, l'outil produit un résultat, puis ce résultat sert à formuler une réponse. Le mode mock fixe les appels à l'avance pour vous permettre d'observer cette chaîne sans dépendre des choix d'un modèle.

Les deux variantes de `dispatch` présentées dans le tutoriel se complètent : conservez la seconde, qui vérifie aussi que les arguments sont un dictionnaire. Comparez les valeurs affichées aux appels du code : `calculate("2+2")` doit retourner `4`, alors que la recherche RAG doit retourner les titres du catalogue. Les noms de fonctions dans la trace constituent les identifiants à vérifier, même si une sortie illustrative emploie un libellé plus général.

En mode mock, mappez trois demandes aux appels attendus. En mode réel, envoyez au modèle les descriptions JSON des outils et exécutez seulement l'appel retourné après validation des arguments.

Sortie obtenue

Les demandes de calcul et de recherche sélectionnent le bon outil et produisent une réponse lisible.

Sortie / exemple

```
request=addition tool=calculator result=42
request=documents tool=search_local hits=2
```

Le mode `--real` envoie uniquement les deux outils déclarés au modèle ; le registre local refuse toujours les noms inconnus.

Code de l'étape

La fonction de `dispatch` est la frontière effective :

Python

```
TOOLS = {"calculate": calculate, "find_course": find_course}

def dispatch(name, arguments):
    if name not in TOOLS:
        raise ValueError(f"outil interdit: {name}")
    if not isinstance(arguments, dict):
        raise ValueError("arguments JSON attendus")
    return TOOLS[name](**arguments)

for call in [
    {"tool": "calculate", "arguments": {"expression": "2+2"}},
    {"tool": "find_course", "arguments": {"topic": "RAG"}},
]:
    print(call["tool"], dispatch(call["tool"], call["arguments"]))
```

Commande de vérification

Terminal · Bash

```
python src/agent.py --mock --demo | tee outputs/demo.txt  
grep -q 'find_course' outputs/demo.txt
```

4. Auditer la trace

Relisez la trace comme une suite d'événements : pour chaque demande, retrouvez soit son résultat, soit le motif de refus. Un résultat numérique correct ne suffit pas à expliquer toute l'exécution si le nom ou les arguments de l'appel ont disparu. Comparez aussi le fichier après deux lancements pour comprendre ce qui relève de la session courante et ce qui est réellement conservé sur disque.

Le format JSONL place un événement JSON sur chaque ligne. Vous pouvez ainsi relire les appels un par un, retrouver leurs arguments et vérifier le résultat sans devoir extraire des informations d'un texte libre. Le statut doit indiquer si l'appel a réussi ou a été refusé ; une réponse finale seule ne suffit pas à reconstituer le parcours.

L'extrait commence par construire les événements en mémoire, puis les écrit dans le fichier. Avant de lancer l'audit, ouvrez celui-ci et comparez les noms d'outils au registre autorisé. Si le compteur d'événements diffère de celui d'un exemple, examinez les appels effectivement réalisés plutôt que d'ajouter une ligne pour obtenir le même total.

Écrivez chaque étape dans `outputs/trace.jsonl`, puis vérifiez qu'aucun outil interdit et aucun argument inattendu n'apparaissent.

Sortie obtenue

La trace permet de reconstituer la décision et montre que l'agent n'a effectué que des lectures ou calculs locaux.

Sortie / exemple

```
trace_ok events=3 forbidden_tools=0 invalid_arguments=0
```

Commande de vérification

Terminal · Bash

```
python src/audit_trace.py outputs/trace.jsonl  
pytest -q
```

Code de l'étape

Chaque appel est enrichi avant d'être écrit dans la trace :

```
Python

import json
from pathlib import Path

trace = []
for name, arguments in [("calculate", {"expression": "2+2"}),
                        ("find_course", {"topic": "RAG"})]:
    result = dispatch(name, arguments)
    trace.append({"tool": name, "arguments": arguments,
                  "status": "ok", "result": result})

Path("outputs/trace.jsonl").write_text(
    "".join(json.dumps(event) + "\n" for event in trace),
    encoding="utf-8",
)
```

Exemples de sorties

Trois demandes sont traitées, une tentative d'outil interdit est bloquée et la trace JSONL est valide.

Dépannage et diagnostics

Validez les types et les bornes des arguments avant l'appel. Le prompt ne constitue pas une frontière de sécurité ; le registre Python doit rester la règle effective.

Vérification finale

- ☐ deux outils déclarés ;
- ☐ outil inconnu bloqué ;
- ☐ arguments contrôlés ;
- ☐ trace complète.

Limites et prolongements

Un agent multi-étapes peut boucler ou coûter cher. Ajoutez un nombre maximal d'appels et une validation humaine pour les actions sensibles.

Références

- [OpenAI — Function calling](#)
- [Python — typing](#)