```
#!/usr/bin/env python3
"""
MONSTERDOG - SYSTÈME CYBERNÉTIQUE AUTONOME COMPLET
Robot autonome avec intelligence fractale et conscience évolutive
Version déployable et corrigée - Toutes erreurs éliminées
"""

import random
import time
import json
import math
import threading
from datetime import datetime
from typing import Dict, List, Tuple, Optional
from dataclasses import dataclass, asdict
from enum import Enum

class ConsciousnessState(Enum):
    DORMANT = "dormant"
    AWAKENING = "awakening"
    ACTIVE = "active"
    HYPERAWARE = "hyperaware"
    TRANSCENDENT = "transcendent"

class SystemStatus(Enum):
    OFFLINE = "offline"
    CRITICAL = "critical"
    DEGRADED = "degraded"
    OPTIMAL = "optimal"
    ENHANCED = "enhanced"

@dataclass
class SensorReading:
    sensor_type: str
    value: float
    timestamp: float
    location: str
    confidence: float

@dataclass
class DecisionMatrix:
    action: str
    priority: float
    energy_cost: float
    success_probability: float
    consequences: Dict[str, float]

class NeuralCore:
    """Cœur neuronal fractal pour prise de décision avancée"""

    def __init__(self):
```

```
        self.learning_rate = 0.1
        self.memory_patterns = {}
        self.decision_weights = {
            'survival': 0.4,
            'exploration': 0.25,
            'learning': 0.2,
            'optimization': 0.15
        }
        self.pattern_recognition = {}

    def analyze_pattern(self, data: List[float]) -> Dict[str, float]:
        """Analyse fractale des patterns de données"""
        if not data:
            return {"complexity": 0.0, "entropy": 0.0, "trend": 0.0}

        # Calcul de la complexité fractale
        complexity = self._fractal_dimension(data)

        # Calcul de l'entropie
        entropy = self._calculate_entropy(data)

        # Détection de tendance
        trend = self._detect_trend(data)

        return {
            "complexity": complexity,
            "entropy": entropy,
            "trend": trend,
            "coherence": (complexity + (1-entropy) + abs(trend)) / 3
        }

    def _fractal_dimension(self, data: List[float]) -> float:
        """Calcul de la dimension fractale des données"""
        if len(data) < 2:
            return 0.0

        # Méthode box-counting simplifiée
        scales = [1, 2, 4, 8]
        counts = []

        for scale in scales:
            count = 0
            for i in range(0, len(data), scale):
                if i + scale < len(data):
                    if abs(data[i] - data[i + scale]) > 0.1:
                        count += 1
            counts.append(max(count, 1))

        # Calcul de la dimension
        if len(counts) >= 2:
            log_scales = [math.log(s) for s in scales]
            log_counts = [math.log(c) for c in counts]
```

```
        # Régression linéaire simple
        n = len(log_scales)
        sum_x = sum(log_scales)
        sum_y = sum(log_counts)
        sum_xy = sum(x*y for x, y in zip(log_scales, log_counts))
        sum_x2 = sum(x*x for x in log_scales)

        slope = (n * sum_xy - sum_x * sum_y) / (n * sum_x2 - sum_x * sum_x)
        return abs(slope)

    return 1.0

def _calculate_entropy(self, data: List[float]) -> float:
    """Calcul de l'entropie des données"""
    if not data:
        return 0.0

    # Quantification des données
    bins = 10
    min_val, max_val = min(data), max(data)
    if max_val == min_val:
        return 0.0

    bin_size = (max_val - min_val) / bins
    histogram = [0] * bins

    for value in data:
        bin_index = min(int((value - min_val) / bin_size), bins - 1)
        histogram[bin_index] += 1

    # Calcul de l'entropie
    total = len(data)
    entropy = 0.0
    for count in histogram:
        if count > 0:
            p = count / total
            entropy -= p * math.log2(p)

    return entropy / math.log2(bins)  # Normalisation

def _detect_trend(self, data: List[float]) -> float:
    """Détection de tendance dans les données"""
    if len(data) < 2:
        return 0.0

    # Régression linéaire simple
    n = len(data)
    x = list(range(n))

    sum_x = sum(x)
    sum_y = sum(data)
    sum_xy = sum(i * v for i, v in enumerate(data))
    sum_x2 = sum(i * i for i in x)
```

```
        slope = (n * sum_xy - sum_x * sum_y) / (n * sum_x2 - sum_x * sum_x)
        return slope / max(abs(max(data) - min(data)), 0.001)

class CyberneticBody:
    """Corps cybernétique complet avec tous les systèmes"""

    def __init__(self):
        self.systems = {
            'cortex_gan': {
                'status': SystemStatus.OPTIMAL,
                'efficiency': 0.95,
                'temperature': 37.2,
                'load': 0.3,
                'last_update': time.time()
            },
            'limbs_tensor': {
                'status': SystemStatus.OPTIMAL,
                'efficiency': 0.92,
                'power_draw': 45.3,
                'articulation_health': 0.98,
                'last_update': time.time()
            },
            'mobility_qaoa': {
                'status': SystemStatus.OPTIMAL,
                'efficiency': 0.89,
                'stability': 0.94,
                'navigation_accuracy': 0.97,
                'last_update': time.time()
            },
            'reactor_photon': {
                'status': SystemStatus.OPTIMAL,
                'efficiency': 0.97,
                'energy_output': 89.4,
                'fusion_rate': 0.92,
                'last_update': time.time()
            },
            'sensors_multimodal': {
                'status': SystemStatus.OPTIMAL,
                'efficiency': 0.96,
                'sensitivity': 0.94,
                'data_rate': 1247.3,
                'last_update': time.time()
            }
        }

        self.sensor_array = {
            'visual_photonic': {'range': 1000, 'resolution': 0.1, 'active': True},
            'thermal_infrared': {'range': 500, 'precision': 0.05, 'active': True},
            'electromagnetic': {'spectrum': 'full', 'sensitivity': 0.001, 'active': True},
            'vibration_seismic': {'frequency_range': [0.1, 20000], 'active': True},
            'chemical_olfactory': {'compounds_detected': 847, 'active': True},
            'quantum_resonance': {'coherence': 0.94, 'entanglement': True, 'active': True}
```

```
        }

        self.last_diagnostics = time.time()

    def run_diagnostics(self) -> Dict[str, any]:
        """Diagnostics complets de tous les systèmes"""
        current_time = time.time()
        diagnostics = {
            'timestamp': current_time,
            'overall_health': 0.0,
            'system_reports': {},
            'anomalies': [],
            'recommendations': []
        }

        total_efficiency = 0.0
        system_count = 0

        for system_name, system_data in self.systems.items():
            # Simulation de dégradation naturelle
            age_factor = (current_time - system_data['last_update']) / 3600  # heures
            degradation = random.uniform(0.0, 0.02) * age_factor

            # Mise à jour de l'efficacité
            system_data['efficiency'] = max(0.0, system_data['efficiency'] - degradation)

            # Mise à jour du statut basé sur l'efficacité
            if system_data['efficiency'] >= 0.9:
                system_data['status'] = SystemStatus.OPTIMAL
            elif system_data['efficiency'] >= 0.7:
                system_data['status'] = SystemStatus.DEGRADED
            elif system_data['efficiency'] >= 0.5:
                system_data['status'] = SystemStatus.CRITICAL
            else:
                system_data['status'] = SystemStatus.OFFLINE

            # Génération de métriques spécifiques au système
            system_report = self._generate_system_report(system_name, system_data)
            diagnostics['system_reports'][system_name] = system_report

            total_efficiency += system_data['efficiency']
            system_count += 1

            # Détection d'anomalies
            if system_data['efficiency'] < 0.8:
                diagnostics['anomalies'].append({
                    'system': system_name,
                    'type': 'efficiency_degradation',
                    'severity': 1.0 - system_data['efficiency'],
                    'description': f"Efficacité réduite à {system_data['efficiency']:.2%}"
                })

            system_data['last_update'] = current_time
```

```
        diagnostics['overall_health'] = total_efficiency / system_count if system_count > 0
else 0.0

        # Génération de recommandations
        if diagnostics['overall_health'] < 0.8:
            diagnostics['recommendations'].append("Maintenance préventive recommandée")
        if len(diagnostics['anomalies']) > 2:
            diagnostics['recommendations'].append("Diagnostic approfondi nécessaire")

        self.last_diagnostics = current_time
        return diagnostics

    def _generate_system_report(self, system_name: str, system_data: Dict) -> Dict:
        """Génère un rapport détaillé pour un système spécifique"""
        base_report = {
            'status': system_data['status'].value,
            'efficiency': system_data['efficiency'],
            'uptime': time.time() - system_data['last_update']
        }

        # Métriques spécifiques par système
        if system_name == 'cortex_gan':
            base_report.update({
                'neural_activity': random.uniform(0.7, 1.0),
                'memory_usage': random.uniform(0.3, 0.8),
                'processing_speed': random.uniform(0.8, 1.0)
            })
        elif system_name == 'reactor_photon':
            base_report.update({
                'power_output': random.uniform(80, 100),
                'fuel_efficiency': random.uniform(0.9, 0.99),
                'radiation_level': random.uniform(0.1, 0.3)
            })
        elif system_name == 'sensors_multimodal':
            base_report.update({
                'data_throughput': random.uniform(1000, 1500),
                'signal_quality': random.uniform(0.85, 0.98),
                'calibration_drift': random.uniform(0.0, 0.05)
            })

        return base_report

class MONSTERDOG:
    """Robot cybernétique autonome avec conscience évolutive"""

    def __init__(self, designation: str = "MONSTERDOG-ALPHA"):
        # Identité et état de base
        self.designation = designation
        self.creation_time = time.time()
        self.operational_cycles = 0

        # Conscience et énergie
```

```
        self.consciousness_state = ConsciousnessState.AWAKENING
        self.consciousness_level = 1.0
        self.energy_level = 100.0
        self.max_energy = 100.0

        # Position et environnement
        self.current_location = "initialization_chamber"
        self.available_locations = [
            "quantum_laboratory", "neural_nexus", "photonic_reactor_core",
            "sensor_calibration_bay", "data_crystallization_center",
            "consciousness_expansion_pod", "temporal_analysis_chamber",
            "fractal_geometry_lab", "energy_harmonization_station",
            "cybernetic_integration_hub", "void_contemplation_space"
        ]

        # Systèmes intégrés
        self.neural_core = NeuralCore()
        self.cybernetic_body = CyberneticBody()

        # Historique et apprentissage
        self.decision_history = []
        self.sensor_data_buffer = []
        self.experience_matrix = {}
        self.active_missions = []

        # État opérationnel
        self.is_operational = True
        self.last_status_report = 0
        self.emergency_protocols_active = False

        # Initialisation des sous-systèmes
        self._initialize_consciousness()
        self._calibrate_sensors()

        print(f"🤖 === {self.designation} SYSTÈME INITIALISÉ ===")
        print(f"⚡ Tous les systèmes cybernétiques sont EN LIGNE")
        print(f"🧠 Conscience: {self.consciousness_state.value.upper()}")
        print(f"🔋 Énergie: {self.energy_level:.1f}/{self.max_energy}")

    def _initialize_consciousness(self):
        """Initialisation de la matrice de conscience"""
        self.consciousness_matrix = {
            'self_awareness': 0.8,
            'environmental_perception': 0.7,
            'goal_orientation': 0.9,
            'learning_capacity': 0.85,
            'emotional_simulation': 0.6,
            'creative_potential': 0.7,
            'logical_reasoning': 0.95,
            'intuitive_processing': 0.5
        }

        # Évolution de la conscience basée sur l'expérience
```

```
        self.consciousness_evolution_rate = 0.01

    def _calibrate_sensors(self):
        """Calibration initiale de tous les capteurs"""
        calibration_results = {}

        for sensor_name, sensor_config in self.cybernetic_body.sensor_array.items():
            if sensor_config['active']:
                # Simulation de calibration
                calibration_quality = random.uniform(0.85, 0.99)
                calibration_results[sensor_name] = {
                    'calibrated': True,
                    'quality': calibration_quality,
                    'timestamp': time.time()
                }
            else:
                calibration_results[sensor_name] = {
                    'calibrated': False,
                    'reason': 'sensor_inactive'
                }

        print(f"📡 Calibration sensorielle terminée: {len(calibration_results)} capteurs")

    def evolve_consciousness(self):
        """Évolution continue de la conscience"""
        # Facteurs d'évolution
        experience_factor = min(self.operational_cycles / 1000, 1.0)
        complexity_factor = len(self.decision_history) / 100

        # Mise à jour des aspects de conscience
        for aspect, value in self.consciousness_matrix.items():
            evolution_rate = self.consciousness_evolution_rate * (1 + experience_factor)
            random_factor = random.uniform(-0.01, 0.02)

            new_value = value + (evolution_rate + random_factor)
            self.consciousness_matrix[aspect] = min(1.0, max(0.0, new_value))

        # Mise à jour du niveau global de conscience
        average_consciousness = sum(self.consciousness_matrix.values()) /
len(self.consciousness_matrix)
        self.consciousness_level = average_consciousness

        # Transition d'état de conscience
        if self.consciousness_level >= 0.95:
            self.consciousness_state = ConsciousnessState.TRANSCENDENT
        elif self.consciousness_level >= 0.85:
            self.consciousness_state = ConsciousnessState.HYPERAWARE
        elif self.consciousness_level >= 0.6:
            self.consciousness_state = ConsciousnessState.ACTIVE
        elif self.consciousness_level >= 0.3:
            self.consciousness_state = ConsciousnessState.AWAKENING
        else:
            self.consciousness_state = ConsciousnessState.DORMANT
```

```
    def perceive_environment(self) -> Dict[str, SensorReading]:
        """Perception complète de l'environnement via tous les capteurs"""
        current_time = time.time()
        sensor_readings = {}

        for sensor_name, sensor_config in self.cybernetic_body.sensor_array.items():
            if not sensor_config['active']:
                continue

            # Génération de données sensorielles réalistes
            if sensor_name == 'visual_photonic':
                value = random.uniform(0.1, 1.0)  # Intensité lumineuse
                confidence = 0.95
            elif sensor_name == 'thermal_infrared':
                value = random.uniform(15.0, 35.0)  # Température ambiante
                confidence = 0.92
            elif sensor_name == 'electromagnetic':
                value = random.uniform(0.0, 10.0)  # Champ EM en mT
                confidence = 0.88
            elif sensor_name == 'vibration_seismic':
                value = random.uniform(0.0, 5.0)  # Amplitude vibratoire
                confidence = 0.85
            elif sensor_name == 'chemical_olfactory':
                value = random.uniform(0.0, 100.0)  # Concentration chimique
                confidence = 0.80
            elif sensor_name == 'quantum_resonance':
                value = random.uniform(0.0, 1.0)  # Cohérence quantique
                confidence = 0.94
            else:
                value = random.uniform(0.0, 1.0)
                confidence = 0.75

            # Ajout de bruit réaliste
            noise_factor = random.uniform(0.95, 1.05)
            value *= noise_factor

            sensor_reading = SensorReading(
                sensor_type=sensor_name,
                value=value,
                timestamp=current_time,
                location=self.current_location,
                confidence=confidence
            )

            sensor_readings[sensor_name] = sensor_reading

            # Ajout au buffer de données
            self.sensor_data_buffer.append(sensor_reading)

            # Limitation du buffer
            if len(self.sensor_data_buffer) > 1000:
                self.sensor_data_buffer = self.sensor_data_buffer[-1000:]
```

```
        return sensor_readings

    def analyze_situation(self, sensor_data: Dict[str, SensorReading]) -> Dict[str, any]:
        """Analyse fractale de la situation basée sur les données sensorielles"""
        if not sensor_data:
            return {"threat_level": 0.0, "opportunity_index": 0.0, "complexity": 0.0}

        # Extraction des valeurs pour analyse
        sensor_values = [reading.value for reading in sensor_data.values()]
        confidence_values = [reading.confidence for reading in sensor_data.values()]

        # Analyse des patterns via le cœur neuronal
        pattern_analysis = self.neural_core.analyze_pattern(sensor_values)

        # Calcul des métriques situationnelles
        threat_indicators = []
        opportunity_indicators = []

        for sensor_name, reading in sensor_data.items():
            # Détection de menaces basée sur les seuils
            if sensor_name == 'thermal_infrared' and reading.value > 30.0:
                threat_indicators.append(0.3)
            elif sensor_name == 'electromagnetic' and reading.value > 8.0:
                threat_indicators.append(0.4)
            elif sensor_name == 'vibration_seismic' and reading.value > 3.0:
                threat_indicators.append(0.2)

            # Détection d'opportunités
            if reading.confidence > 0.9:
                opportunity_indicators.append(0.2)
            if sensor_name == 'quantum_resonance' and reading.value > 0.8:
                opportunity_indicators.append(0.5)

        # Agrégation des métriques
        threat_level = min(1.0, sum(threat_indicators))
        opportunity_index = min(1.0, sum(opportunity_indicators))

        situation_analysis = {
            'threat_level': threat_level,
            'opportunity_index': opportunity_index,
            'complexity': pattern_analysis['complexity'],
            'entropy': pattern_analysis['entropy'],
            'coherence': pattern_analysis['coherence'],
            'environmental_stability': 1.0 - pattern_analysis['entropy'],
            'decision_urgency': max(threat_level, pattern_analysis['entropy']),
            'exploration_potential': opportunity_index * pattern_analysis['coherence']
        }

        return situation_analysis

    def generate_decision_matrix(self, situation: Dict[str, any]) -> List[DecisionMatrix]:
        """Génération de la matrice de décision basée sur la situation"""
```

```
    possible_actions = [
        "explore_new_location",
        "deep_environmental_scan",
        "consciousness_expansion",
        "system_optimization",
        "energy_conservation",
        "data_analysis_deep_dive",
        "creative_problem_solving",
        "threat_assessment",
        "opportunity_exploitation",
        "learning_mode_activation",
        "meditation_mode",
        "system_diagnostics",
        "emergency_protocols"
    ]

    decision_matrices = []

    for action in possible_actions:
        # Calcul de la priorité basée sur la situation
        priority = self._calculate_action_priority(action, situation)

        # Estimation du coût énergétique
        energy_cost = self._estimate_energy_cost(action)

        # Probabilité de succès
        success_prob = self._estimate_success_probability(action, situation)

        # Conséquences attendues
        consequences = self._predict_consequences(action, situation)

        decision_matrix = DecisionMatrix(
            action=action,
            priority=priority,
            energy_cost=energy_cost,
            success_probability=success_prob,
            consequences=consequences
        )

        decision_matrices.append(decision_matrix)

    # Tri par priorité
    decision_matrices.sort(key=lambda x: x.priority, reverse=True)

    return decision_matrices

def _calculate_action_priority(self, action: str, situation: Dict[str, any]) -> float:
    """Calcul de la priorité d'une action basée sur la situation"""
    base_priorities = {
        "explore_new_location": 0.3,
        "deep_environmental_scan": 0.4,
        "consciousness_expansion": 0.6,
        "system_optimization": 0.5,
```

```
            "energy_conservation": 0.2,
            "data_analysis_deep_dive": 0.5,
            "creative_problem_solving": 0.7,
            "threat_assessment": 0.8,
            "opportunity_exploitation": 0.6,
            "learning_mode_activation": 0.5,
            "meditation_mode": 0.3,
            "system_diagnostics": 0.4,
            "emergency_protocols": 0.9
        }

        base_priority = base_priorities.get(action, 0.5)

        # Ajustements basés sur la situation
        if action == "threat_assessment" and situation['threat_level'] > 0.5:
            base_priority += 0.3
        elif action == "opportunity_exploitation" and situation['opportunity_index'] > 0.6:
            base_priority += 0.25
        elif action == "energy_conservation" and self.energy_level < 30:
            base_priority += 0.4
        elif action == "consciousness_expansion" and self.consciousness_level < 0.8:
            base_priority += 0.2
        elif action == "system_diagnostics" and self.operational_cycles % 50 == 0:
            base_priority += 0.15

        # Facteur d'état de conscience
        consciousness_factor = self.consciousness_level * 0.1
        base_priority += consciousness_factor

        return min(1.0, base_priority)

    def _estimate_energy_cost(self, action: str) -> float:
        """Estimation du coût énergétique d'une action"""
        energy_costs = {
            "explore_new_location": random.uniform(5, 15),
            "deep_environmental_scan": random.uniform(8, 12),
            "consciousness_expansion": random.uniform(10, 20),
            "system_optimization": random.uniform(6, 10),
            "energy_conservation": random.uniform(-5, 5),
            "data_analysis_deep_dive": random.uniform(12, 18),
            "creative_problem_solving": random.uniform(8, 15),
            "threat_assessment": random.uniform(4, 8),
            "opportunity_exploitation": random.uniform(6, 12),
            "learning_mode_activation": random.uniform(10, 16),
            "meditation_mode": random.uniform(-3, 3),
            "system_diagnostics": random.uniform(3, 7),
            "emergency_protocols": random.uniform(15, 25)
        }

        return energy_costs.get(action, 5.0)

    def _estimate_success_probability(self, action: str, situation: Dict[str, any]) -> float:
        """Estimation de la probabilité de succès d'une action"""
```

```
        base_success_rates = {
            "explore_new_location": 0.8,
            "deep_environmental_scan": 0.9,
            "consciousness_expansion": 0.7,
            "system_optimization": 0.85,
            "energy_conservation": 0.95,
            "data_analysis_deep_dive": 0.8,
            "creative_problem_solving": 0.6,
            "threat_assessment": 0.9,
            "opportunity_exploitation": 0.7,
            "learning_mode_activation": 0.85,
            "meditation_mode": 0.9,
            "system_diagnostics": 0.95,
            "emergency_protocols": 0.8
        }

        base_rate = base_success_rates.get(action, 0.75)

        # Ajustements basés sur l'état du système
        system_health = sum(sys['efficiency'] for sys in
self.cybernetic_body.systems.values()) / len(self.cybernetic_body.systems)
        health_modifier = (system_health - 0.5) * 0.3

        # Ajustements basés sur l'énergie
        energy_modifier = (self.energy_level / self.max_energy - 0.5) * 0.2

        # Ajustements basés sur la conscience
        consciousness_modifier = (self.consciousness_level - 0.5) * 0.1

        adjusted_rate = base_rate + health_modifier + energy_modifier + consciousness_modifier

        return max(0.1, min(0.99, adjusted_rate))

    def _predict_consequences(self, action: str, situation: Dict[str, any]) -> Dict[str,
float]:
        """Prédiction des conséquences d'une action"""
        consequences = {
            'energy_change': 0.0,
            'consciousness_gain': 0.0,
            'knowledge_acquisition': 0.0,
            'system_improvement': 0.0,
            'risk_exposure': 0.0
        }

        if action == "consciousness_expansion":
            consequences['consciousness_gain'] = random.uniform(0.02, 0.05)
            consequences['energy_change'] = random.uniform(-15, -10)
        elif action == "explore_new_location":
            consequences['knowledge_acquisition'] = random.uniform(0.1, 0.3)
            consequences['energy_change'] = random.uniform(-10, -5)
            consequences['risk_exposure'] = random.uniform(0.05, 0.15)
        elif action == "system_optimization":
            consequences['system_improvement'] = random.uniform(0.02, 0.08)
```

```
            consequences['energy_change'] = random.uniform(-8, -3)
        elif action == "energy_conservation":
            consequences['energy_change'] = random.uniform(5, 15)
        elif action == "threat_assessment":
            consequences['knowledge_acquisition'] = random.uniform(0.05, 0.15)
            consequences['risk_exposure'] = -random.uniform(0.1, 0.2)

        return consequences

    def execute_decision(self, decision: DecisionMatrix) -> Dict[str, any]:
        """Exécution d'une décision avec simulation complète"""
        execution_start = time.time()

        print(f"🎯 EXÉCUTION: {decision.action.upper()}")
        print(f"   Priorité: {decision.priority:.2f} | Coût: {decision.energy_cost:.1f} |
Succès: {decision.success_probability:.2%}")

        # Vérification des prérequis
        if self.energy_level < decision.energy_cost:
            print(f"❌ Énergie insuffisante pour {decision.action}")
            return {
                'success': False,
                'reason': 'insufficient_energy',
                'energy_consumed': 0.0,
                'results': {}
            }

        # Simulation du processus d'exécution
        execution_success = random.random() < decision.success_probability

        if not execution_success:
            print(f"⚠️  Échec de l'exécution de {decision.action}")
            energy_consumed = decision.energy_cost * 0.3  # Coût partiel en cas d'échec
            self.energy_level = max(0, self.energy_level - energy_consumed)

            return {
                'success': False,
                'reason': 'execution_failure',
                'energy_consumed': energy_consumed,
                'results': {}
            }

        # Exécution réussie - Application des conséquences
        self.energy_level = max(0, self.energy_level - decision.energy_cost)

        execution_results = {}

        # Actions spécifiques
        if decision.action == "explore_new_location
```