Робота з даними у Python

Робота з даними у Python включає в себе використання різних бібліотек та інструментів для обробки, аналізу та візуалізації даних. Ось деякі ключові бібліотеки та інструменти для роботи з даними у Python:

1. **Pandas:** Pandas - це потужна бібліотека для обробки та аналізу даних. Вона надає структури даних, такі як DataFrame, які дозволяють зручно працювати з табличними даними.

```
import pandas as pd

# Завантаження даних з CSV файлу
df = pd.read_csv('назва_файлу.csv')

# Виведення перших рядків DataFrame
print(df.head())
```

2. **NumPy:** NumPy - це бібліотека для виконання операцій над масивами та матрицями. Вона дозволяє ефективно виконувати числові операції.

```
import numpy as np

# Створення масиву
arr = np.array([1, 2, 3, 4, 5])

# Математичні операції з масивами
squared_arr = np.square(arr)
```

3. **Matplotlib та Seaborn:** Ці бібліотеки використовуються для візуалізації даних.

```
import matplotlib.pyplot as plt
import seaborn as sns

# Гістограма
sns.histplot(df['column_name'])
plt.show()
```

4. **Scikit-learn:** Scikit-learn - це бібліотека для машинного навчання, але вона також надає інструменти для попередньої обробки даних.

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Розділення даних на тренувальний та тестовий набори
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Нормалізація даних
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
```

5. **Jupyter Notebooks:** Jupyter Notebooks - це інтерактивне середовище для розробки, яке дозволяє об'єднувати код, текст та візуалізацію у документі.

```
jupyter notebook
```

Ці інструменти допоможуть вам ефективно працювати з даними у Python та виконувати різноманітні завдання від обробки до аналізу та візуалізації.

Методи роботи з даними

1. **`zip()`**: Зипує (об'єднує) елементи з двох або більше ітерабельних об'єктів.

```
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']

zipped = zip(list1, list2)
result = list(zipped)
# Результат: [(1, 'a'), (2, 'b'), (3, 'c')]
```

2. **`sorted()`**: Сортує ітерабельний об'єкт (список, кортеж, рядок) та повертає новий список.

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
sorted_numbers = sorted(numbers)
# Результат: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
```

3. **`enumerate()`**: Повертає ітератор, який містить пари індексу та відповідного елементу.

```
words = ['apple', 'banana', 'cherry']
for index, value in enumerate(words):
    print(f"Index: {index}, Value: {value}")
# Результат:
# Index: 0, Value: apple
# Index: 1, Value: banana
# Index: 2, Value: cherry
```

4. **`filter()`**: Фільтрує елементи ітерабельного об'єкта за допомогою заданої функції.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
filtered_numbers = filter(lambda x: x % 2 == 0, numbers)
# Результат: [2, 4, 6, 8, 10]
```

5. **`map()`**: Застосовує задану функцію до кожного елементу ітерабельного об'єкта.

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(lambda x: x**2, numbers)
# Результат: [1, 4, 9, 16, 25]
```

6. **`reduce()`**: Застосовує задану функцію активатора до елементів ітерабельного об'єкта для отримання одного значення.

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
# Результат: 120 (1 * 2 * 3 * 4 * 5)
```

7. **`any()` та `all()`**: Перевіряють, чи є хоча б один елемент в ітерабельному об'єкті `True` або чи всі елементи `True`, відповідно.

```
bool_list = [True, False, True, True]
any_result = any(bool_list)  # True
all_result = all(bool_list)  # False
```

8. **`min()` та `max()`**: Знаходять мінімальний або максимальний елемент в ітерабельному об'єкті.

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
min_number = min(numbers)  # 1
max_number = max(numbers)  # 9
```

9. **`zip_longest()`**: Зипує два або більше ітерабельних об'єктів, але додає значення `fillvalue` для пар, де один ітерабельний об'єкт коротший за інший.

```
from itertools import zip_longest

list1 = [1, 2, 3]
list2 = ['a', 'b']

zipped = zip_longest(list1, list2, fillvalue='N/A')
result = list(zipped)
# Результат: [(1, 'a'), (2, 'b'), (3, 'N/A')]
```

itertools

`itertools` - це модуль у стандартній бібліотеці Python, який містить ряд корисних ітераторів та функцій для роботи з ітерацією та комбінаторикою. Давайте розглянемо деякі з найбільш використовуваних інструментів з `itertools`:

1. **`count(start, step)`**: Генерує нескінченну арифметичну прогресію, починаючи з `start` та з кроком `step`.

```
from itertools import count

for i in count(start=1, step=2):
    print(i)
# Виведе: 1, 3, 5, 7, 9, ...
```

2. **`cycle(iterable)`**: Зациклює ітерабельний об'єкт, створюючи нескінченну послідовність.

```
from itertools import cycle
```

```
colors = cycle(['red', 'green', 'blue'])
for color in colors:
    print(color)
# Виведе: red, green, blue, red, green, blue, ...
```

3. **`repeat(element, times)`**: Повертає нескінчений ітератор, який повертає заданий елемент `times` разів або нескінченно, якщо `times` не вказано.

```
from itertools import repeat

repeated_numbers = repeat(2, times=3)
result = list(repeated_numbers)
# Результат: [2, 2, 2]
```

4. **`chain(iterable1, iterable2, ...)`**: Об'єднує декілька ітерабельних об'єктів у один послідовний ітератор.

```
from itertools import chain

list1 = [1, 2, 3]
tuple1 = ('a', 'b', 'c')
combined = chain(list1, tuple1)
result = list(combined)
# Результат: [1, 2, 3, 'a', 'b', 'c']
```

5. **`combinations(iterable, r)` та `permutations(iterable, r)`**: Генерують всі можливі комбінації або перестановки заданого розміру `r` з ітерабельного об'єкта.

```
from itertools import combinations, permutations

items = ['a', 'b', 'c']
all_combinations = combinations(items, 2)
all_permutations = permutations(items, 2)

print(list(all_combinations))
# Результат: [('a', 'b'), ('a', 'c'), ('b', 'c')]

print(list(all_permutations))
# Результат: [('a', 'b'), ('a', 'c'), ('b', 'a'), ('b', 'c'), ('c',
'a'), ('c', 'b')]
```

Це лише кілька прикладів функцій та ітераторів, які пропонує модуль `itertools`. Ці інструменти допомагають вирішувати завдання, пов'язані з ітерацією та комбінаторикою, і роблять роботу з ітерацією більш ефективною та зручною.

functools

`functools` - це модуль у стандартній бібліотеці Python, який надає ряд корисних інструментів для роботи з функціями. Основна мета `functools` - забезпечити функціональність вищого порядку в Python, таку як мемоїзація, каррінг, редукція та інші. Ось деякі з основних інструментів, які надає `functools`:

1. **`functools.partial(func, *args, **kwargs)`**: Застосовує часткове застосування аргументів до функції, що дозволяє створювати нові функції на основі існуючих з частково визначеними параметрами.

```
from functools import partial

def power(base, exponent):
    return base ** exponent

square = partial(power, exponent=2)
cube = partial(power, exponent=3)

print(square(4))  # Результат: 16
print(cube(2))    # Результат: 8
```

2. **`functools.reduce(func, iterable[, initializer])`**: Застосовує бінарну функцію `func` до елементів ітерабельного об'єкта, зліва направо, з обов'язковим ініціалізатором, якщо він вказаний.

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]
result = reduce(lambda x, y: x * y, numbers)
# Результат: 120 (1 * 2 * 3 * 4 * 5)
```

3. **`functools.lru_cache(maxsize=None)`**: Реалізує мемоізацію (кешування результатів попередніх викликів функції) з обмеженням кешу за розміром.

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(5))  # Результат: 5
```

4. **`functools.wraps(wrapped)`**: Декоратор для збереження метаданих функції при використанні декораторів.

```
from functools import wraps

def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Calling function")
        result = func(*args, **kwargs)
        print("Function call complete")
        return result
    return wrapper

@my_decorator
def example_function():
    """Docstring of example_function."""
    print("Inside function")
```

```
example_function()
print(example_function.__name__)  # Результат: example_function
print(example_function.__doc__)   # Результат: Docstring of
example_function.
```

Ці інструменти забезпечують додаткову функціональність при роботі з функціями в Python, допомагаючи управляти аргументами, редукцією, мемоїзацією та збереженням метаданих при використанні декораторів.