ММП 2025/2026

Викладач Канцедал Георгій Олегович

Що було б добре щоб ви памятали після лекції?

- Універсальність Python проявляється в трьох основних парадигмах: імперативному, об’єктно-орієнтованому та функціональному програмуванні.
- Імперативне програмування наголошує на покрокових інструкціях, ООП зосереджується на об'єктах і класах, а функціональне – на обчисленні функцій.
- Python забезпечує безшовну інтеграцію цих парадигм, надаючи гнучкість у розробці програмного забезпечення.
- Гібридні підходи поєднують парадигми, використовуючи їхні сильні сторони для створення ефективного та виразного коду.
- Важливо ретельно продумати поєднання різних парадигм, щоб зберегти узгодженість і читабельність коду.

Імперативна парадигма

- Ця парадигма зосереджується на описі того, як працює програма, використовуючи оператори, що змінюють її стан. Це найбільш поширений і традиційний спосіб написання коду, де ви задаєте послідовність кроків, які комп’ютер має виконати для розв’язання задачі.

Воно характеризується:

- Послідовністю операторів
- Використанням змінних для збереження стану
- Циклами та умовними конструкціями
- Зміною даних безпосередньо в місці їх зберігання

Імперативна парадигма

Переваги імперативного програмування:

- **Інтуїтивність:** Його часто легше зрозуміти та писати, особливо для початківців.
- **Ефективність для певних завдань:** Деякі проблеми природно вирішуються покроковими методами.
- **Прямий контроль:** Ви маєте явний контроль над станом програми на кожному кроці.

Недоліки:

- **Побічні ефекти:** Функції можуть змінювати стан за межами своєї області видимості, що може призводити до непередбачуваної поведінки.
- **Складність паралелізації:** Через залежність від стану може бути важко виконувати операції одночасно.
- **Прийдеться багато писати:** Деякі операції можуть вимагати більше коду порівняно з функціональними підходами.

Імперативна парадигма

IP[y]: Notebook spectrogram Last Checkpoint: a few seconds ago (autosaved) IPython (Python 3)

File Edit View Insert Cell Kernel Help

Code Cell Toolbar: None

Simple spectral analysis

An illustration of the Discrete Fourier Transform using windowing, to reveal the frequency content of a sound signal.

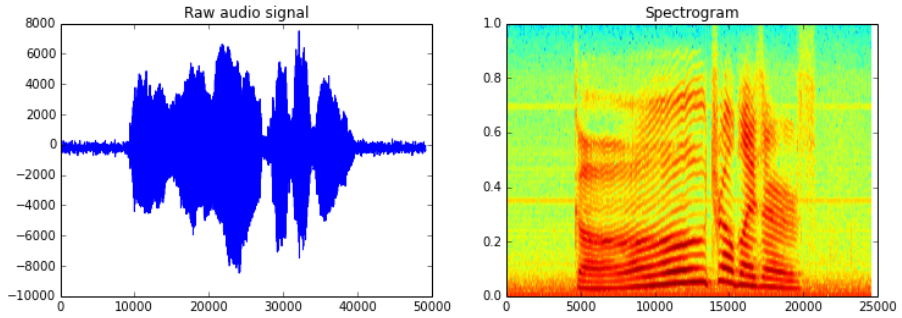
$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \qquad k = 0, \ldots, N-1$$

We begin by loading a datafile using SciPy's audio file support:

In [1]:
```
from scipy.io import wavfile
rate, x = wavfile.read('test_mono.wav')
```

And we can easily view its spectral structure using matplotlib's builtin `specgram` routine:

In [2]:
```
%matplotlib inline
from matplotlib import pyplot as plt
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
ax1.plot(x); ax1.set_title('Raw audio signal')
ax2.specgram(x); ax2.set_title('Spectrogram');
```



Функціональне парадигма

- Функціональне програмування в Python зосереджується на використанні функцій для перетворення даних, уникаючи змінюваного стану та побічних ефектів.

Ключові характеристики:

- Незмінність (immutability)
- Чисті функції (pure functions)
- Функції першого класу та функції вищого порядку
- Рекурсія замість циклів

Функціональне парадигма

Переваги функціонального програмування:

- **Передбачуваність:** Чисті функції легше тестувати та налагоджувати.
- **Паралельність:** Відсутність спільного стану спрощує паралельне виконання.
- **Модульність:** Функції легко поєднувати та повторно використовувати.

Недоліки:

- **Порог входу:** Може бути складним для розробників, звиклих до імперативного стилю.
- **Витрати на продуктивність:** Деякі функціональні конструкції можуть незначно знижувати продуктивність.
- **Менш інтуїтивне для деяких завдань:** Деякі проблеми природніше виражаються в імперативному стилі.

Функціональне парадигма

```python
def factorial_functional(n):
    return 1 if n == 0 else n * factorial_functional(n - 1)

# Usage
print(factorial_functional(5))  # Output: 120
```

```python
# Map: Apply a function to every item in an iterable
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))
print(squared)  # Output: [1, 4, 9, 16, 25]

# Filter: Create a list of elements for which a function returns True
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Output: [2, 4]
```

```python
@pytest.mark.parametrize('dates', ['2022-01-31'])
@pytest.mark.parametrize('folders', [FoldersStructure.players, FoldersStructure.payments])
def test_folder_in_bucket_doesnt_exist(self, dates: str, folders: Tuple[str]):
    src_files_validator = SourceFilesValidator(
        bucket='staging-olg-in', root_dir='test/negative_flows/empty_folder'
    )
    s3_key = src_files_validator.build_resulting_path(dates, folders)
    bucket_contents = src_files_validator.list_objects_in_s3_folder(s3_key)
    src_files_validator.validate_folder(dates, folders, bucket_contents)

    assert len(src_files_validator.get_specified_problems(EmptyFolder)) == 1
```

Порівняння

```python
def sum_of_even_squares_imperative(numbers):
    result = 0
    for num in numbers:
        if num % 2 == 0:
            result += num ** 2
    return result

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(sum_of_even_squares_imperative(numbers))  # Output: 220
```

```python
from functools import reduce

def sum_of_squares_of_even_numbers(numbers):
    # Filter even numbers from the list
    even_numbers = filter(lambda x: x % 2 == 0, numbers)

    # Map each even number to its square
    squares = map(lambda x: x ** 2, even_numbers)

    # Reduce the list of squares to their sum
    sum_of_squares = reduce(lambda x, y: x + y, squares, 0)

    return sum_of_squares

# Example usage
numbers = [1, 2, 3, 4, 5, 6]
result = sum_of_squares_of_even_numbers(numbers)
print(f"The sum of squares of even numbers in the list is: {result}")
```

Кінець