

Pythonic이란 무엇인가?

Zen of Python

Beautiful is better than ugly. 아름다움이 추한 것보다 낫다.

Explicit is better than implicit. 명확함이 함축된 것보다 낫다.

Simple is better than complex. 단순함이 복잡한 것보다 낫다.

Complex is better than complicated. 복잡함이 난해한 것보다 낫다.

Flat is better than nested. 단조로움이 중첩된 것보다 낫다.

Sparse is better than dense. 여유로움이 밀집된 것보다 낫다.

.
.br/>.

<https://wikedocs.net/7907>

Zen of Python

Beautiful is better than ugly. 아름다움이 추한 것보다 낫다.

Explicit is better than implicit. 명확함이 함축된 것보다 낫다.

.

간결하고 가독성이 좋은 코드!

<https://wikedocs.net/7907>

연산자 오버로딩

- 연산자 재정의

연산자 오버로딩

- 연산자 재정의

1. PyTorch
2. LangChain

Python

```
In [1]: 40 + 2
```

```
Out[1]: 42
```

```
In [2]: "lorem " + "ipsum"
```

```
Out[2]: 'lorem ipsum'
```

```
In [3]: [1, 2, 3] + [4, 5, 6]
```

```
Out[3]: [1, 2, 3, 4, 5, 6]
```

```
In [4]: import torch
```

```
In [5]: torch.Tensor([1, 2, 3]) + torch.Tensor([4, 5, 6])
```

```
Out[5]: tensor([5., 7., 9.])
```

Python

```
In [1]: 40 + 2
```

```
Out[1]: 42
```

```
In [2]: "lorem " + "ipsum"
```

```
Out[2]: 'lorem ipsum'
```

```
In [3]: [1, 2, 3] + [4, 5, 6]
```

```
Out[3]: [1, 2, 3, 4, 5, 6]
```

```
In [4]: import torch
```

```
In [5]: torch.Tensor([1, 2, 3]) + torch.Tensor([4, 5, 6])
```

```
Out[5]: tensor([5., 7., 9.])
```

같은 + 연산자 인데 **타입에 따라** 동작이 다르다

Python

```
[1, 2, 3] + [4, 5, 6]
```


Python

```
[1, 2, 3] + [4, 5, 6]
```

```
[1, 2, 3].__add__([4, 5, 6])
```

Python

```
[1, 2, 3] + [4, 5, 6]
```

```
[1, 2, 3].__add__([4, 5, 6])
```

객체마다 `__add__()` 를 다르게 구현 (매직 메서드)

PyTorch

향해
PLUS

텐서 덧셈 직접 구현해보기!

- [PyTorch: An Imperative Style, High-Performance Deep Learning Library](#)
 - Tensorflow 및 기타 라이브러리와 비교
 - PyTorch는 성능과 사용성을 모두 잡은 라이브러리

```
import torch
torch.Tensor([1, 2, 3]) + torch.Tensor([1, 2, 3])

import tensorflow as tf
tf.add(tf.constant([1, 2, 3]), tf.constant([1, 2, 3]))
```

- [PyTorch: An Imperative Style, High-Performance Deep Learning Library](#)
 - Tensorflow 및 기타 라이브러리와 비교
 - PyTorch는 성능과 사용성을 모두 잡은 라이브러리

```
import torch
torch.Tensor([1, 2, 3]) + torch.Tensor([1, 2, 3])

import tensorflow as tf
tf.add(tf.constant([1, 2, 3]), tf.constant([1, 2, 3]))
```

```
tf.constant([1, 2, 3]) + tf.constant([1, 2, 3])
```

- [PyTorch: An Imperative Style, High-Performance Deep Learning Library](#)
 - Tensorflow 및 기타 라이브러리와 비교
 - PyTorch는 성능과 사용성을 모두 잡은 라이브러리



Andrej Karpathy ✓

@karpathy



I've been using PyTorch a few months now and I've never felt better. I have more energy. My skin is clearer. My eye sight has improved.

게시물 번역하기

오전 3:56 · 2017년 5월 27일

424 재게시 **86** 인용 **2,006** 마음에 들어요 **53** 북마크

LangChain

- LangChain pipe [교안 링크](#)

위의 일련의 과정은 다음과 같이 chain의 형태로 쉽게 구현할 수도 있습니다:

```
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough

rag_chain = (
    {"context": retriever | format_docs, "question": RunnablePassthrough()}
    | prompt
    | llm
    | StrOutputParser()
)
print(rag_chain.invoke(user_msg))
```

LangChain

- LangChain pipe [교안 링크](#)

위의 일련의 과정은 다음과 같이 chain의 형태로 쉽게 구현할 수도 있습니다:

```

from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough

rag_chain = (
    {"context": retriever | format_docs, "question": RunnablePassthrough()}
    | prompt
    | llm
    | StrOutputParser()
)
print(rag_chain.invoke(user_msg))
  
```

- 원래 용도는 논리 연산

False		False	#	False
False		True	#	True
True		False	#	True
True		True	#	True

- __or__ 메서드를 구현하면 동작을 재정의 할 수 있음.

```
def inc(x: int) → int:  
    return x + 1  
  
def square(x: int) → int:  
    return x * x  
  
x1 = 1  
x2: int = inc(x1)      # 2  
x3: int = inc(x2)      # 3  
x4: int = inc(x3)      # 4  
x5: int = square(x4)   # 16  
  
x5                      # 16
```

- __or__ 메서드를 구현하면 동작을 재정의 할 수 있음.

```
def inc(x: int) → int:
    return x + 1

def square(x: int) → int:
    return x * x

x1 = 1
x2: int = inc(x1)      # 2
x3: int = inc(x2)      # 3
x4: int = inc(x3)      # 4
x5: int = square(x4)   # 16

x5                      # 16
```



```
@pipe_decorator
def inc(x: int) → int:
    return x + 1

@pipe_decorator
def square(x: int) → int:
    return x * x

# square(inc(inc(inc(1))))    # 16
1 | inc | inc | inc | square # 16
```


```
@pipe_decorator  
def load_data(dataset):  
    pass
```

```
@pipe_decorator  
def preprocess(loaded_dataset):  
    pass
```

```
@pipe_decorator  
def train(processed_dataset):  
    pass
```

```
@pipe_decorator  
def evaluate(model):  
    pass
```

```
def train_pipeline(dataset):  
    return dataset  
    |  
    | load_data  
    |  
    | preprocess  
    |  
    | train  
    |  
    | evaluate
```



The diagram illustrates the relationship between individual pipeline steps and a combined pipeline function. Four red arrows originate from the left panel and point to the right panel:

- The first arrow points from the `load_data` function to the `load_data` step in the `train_pipeline` function.
- The second arrow points from the `preprocess` function to the `preprocess` step in the `train_pipeline` function.
- The third arrow points from the `train` function to the `train` step in the `train_pipeline` function.
- The fourth arrow points from the `evaluate` function to the `evaluate` step in the `train_pipeline` function.

감사합니다!

<https://github.com/jw3215/hh-plus-ai-1-presentation/>