

**深入淺出 Python**  
**A Tutorial on Python**  
**2023.07.04**  
**Wei-Chen Liao**  
**@Deep Learning**

# Outline

- **Environment**
- **Built-in Functions**
- **Packing and Unpacking**
- **Class**

# Outline

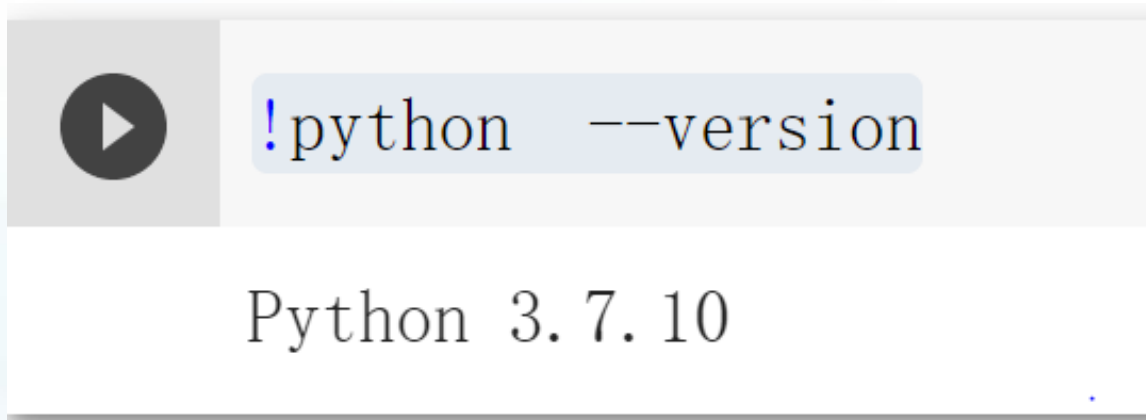
- **Environment**
- **Built-in Functions**
- **Packing and Unpacking**
- **Class**

# Environments

- (NYCU) We'll provide workstation
- (Local) Install Python3
- (Local / Remote) Jupyter Notebook
- (Cloud) Google Colab

# Python version

- `python >= 3.6`
- Command: `python -V`
- Colab: `!python --version`



```
!python --version
```

Python 3.7.10

# Outline

- **Environment**
- **Built-in Functions**
- **Packing and Unpacking**
- **Class**

# Built-in Functions

- Reference: [Built-in Functions — Python 3.9.2 documentation](#)

# Example

```
data = [1, 3, 2]
print(max(data)) # 3
print(min(data)) # 1
print(sum(data)) # 6
print(len(data)) # 3

# Example: loss less than threshold
dones = [True, False, True, True]
all_done = all(dones) # False
any_done = any(dones) # True
```



# F-string

- In front of the string, add an “f”
- Within strings, enclose variables in curly braces “{ }”

```
epoch, loss, acc = 600, 1.2345, 0.87654321
print(f'Epoch: {epoch}, loss: {loss}, accuracy: {acc}')
#Epoch:600, loss:1.2345, accuracy:0.87654321
print(f'Epoch: {epoch:4d}, loss: {loss:5.2f}, accuracy: {acc:.2%}')
#Epoch:  600, loss:  1.23, accuracy:87.65%
```

```
Epoch:600, loss:1.2345, accuracy:0.87654321
```

```
Epoch: 600, loss: 1.23, accuracy:87.65%
```

# String `format()` Method

```
▶ epoch, loss = 600, 0.12345
a = 'Epoch {epoch}, loss: {loss}'
print(a)
# Epoch {epoch}, loss: {loss}
print(a.format(epoch=epoch, loss=loss))
# Epoch 600, loss: 0.12345
b = 'Epoch {:4d}, loss: {:.2f}'
print(b)
# Epoch {:4d}, loss: {:.2f}
print(b.format(epoch, loss))
# Epoch 600, loss: 0.12
```

```
Epoch {epoch}, loss: {loss}
Epoch 600, loss: 0.12345
Epoch {:4d}, loss: {:.2f}
Epoch 600, loss: 0.12
```

# Enumerate

iterable: list, tuple, dict, ...

- `enumerate(iterable, start=0)` `>>> print(list(enumerate(['A', 'B', 'C'])))`  
`[(0, 'A'), (1, 'B'), (2, 'C')]`

Img1

Img2

Img3

```
data = [Img(), Img(), Img()]  
for i, x in enumerate(data):  
    print(i, x)
```

0    Img1

1    Img2

2    Img3

# Open

```
# mode 'r': read (default)
# mode 'w': write
# mode 'a': append
f = open('test.txt', 'w')
f.write('zzz')
f.close()
```

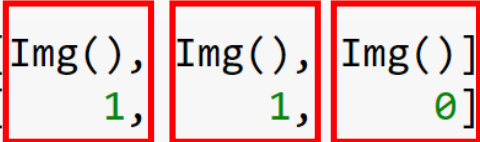
Writing in a more simple manner:

```
with open('test.txt', 'w') as f:
    print('zzz', file=f)
    f.write('zzz')
```

# zip

```
images = [Img(), Img(), Img()]  
labels = [1, 1, 0]
```

```
for img, label in zip(images, labels):  
    do_something(image, label)
```

```
images = [Img(), Img(), Img()]  
labels = [
```

Combined with enumerate:

```
for i, (img, label) in enumerate(zip(images, labels), start=1):  
    do_something(image, label)
```

# map

map([function](#), iterable, ...) is usually mixed with a list to produce a list

```
raw_data = ['1', '2', '3']  
data = list(map(int, raw_data))  
# [1, 2, 3]
```

# Lambda Function

```
def f(x):  
    return x ** 3 + 3 * (x ** 2) + 1  
  
# Lambda function  
f2 = lambda x : x ** 3 + 3 * (x ** 2) + 1  
  
print(f(-1), f2(-1))
```

Lambda Function can combine with “map”

```
raw_data = ['1', '2', '3']  
data = list(map(lambda n : n + 'x', raw_data))  
# ['1x', '2x', '3x']
```

# Default Argument and Positional Argument

```
def f(x=1, y=1):  
    return x ** 3 + y ** 3
```

```
print(f(), f(0), f(0, 2))  
print(f(y=3)) # 指定傳參數
```

```
# BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True)  
bn = BatchNorm2d(2)  
bn = BatchNorm2d(num_features=2)  
bn = BatchNorm2d(2, track_running_stats=False)
```



# List Comprehensions

```
# Original:  
squares = []  
for x in range(10):  
    squares.append(x ** 2)  
  
# List comprehension:  
squares = [x ** 2 for x in range(10)]
```

# List Comprehensions: Example

```
def preprocess(x) :  
    # do something  
    return x  
  
raw_data = ['This is demo.', 'For demo!']  
  
input_data = [preprocess(x) for x in raw_data]  
  
print([w.shape for w in weights])  
  
data = [1, 2, 3]  
data_tensor = [Tensor([x]) for x in data]
```

# List Comprehensions: Example

```
raw_data = [(1, 2), (2, 2), (3, 4)]  
data = [preprocess(x, y) for x, y in raw_data if x != y]
```

```
char_to_index = {'a': 0, 'b': 1}  
index_to_char = {v: k for k, v in char_to_index.items()}
```

# Outline

- **Environment**
- **Built-in Functions**
- **Packing and Unpacking**
- **Class**

# \*args

Based on the direction of data flow, it can be used as Packing or Unpacking.

“\*” a packing effect:

```
def print_repeat(num, *content):  
    for _ in range(num):  
        print(content) ← Tuple: ('a','b','c')  
  
x = print_repeat(3, 'a', 'b', 'c')
```

```
('a', 'b', 'c')  
( 'a', 'b', 'c')  
( 'a', 'b', 'c')
```

## \*args : example

“\*” a packing effect:

```
def p_norm(*v, p):  
    ret = 0  
    for k in v:  
        ret += k ** p  
    return ret ** (1 / p)
```

keyword-only argument

```
print(p_norm(1, 2, 3, p=2))
```

v: (1,2,3)  
p: 2

3.7416573867739413

# Iterable Unpacking

- tuple, list, str
- Underlining “  ” indicates that the variable is ignored

```
_, value = net(input)

dim = (4096, 3, 80, 80) # [N, C, H, W]
batch_size, _, height, width = dim
```

- Use with "\*" (used as a packing indicator)

```
_, _, *size = dim
# size == [80, 80]

batch_size, *_ = dim
# batch_size == 4096
```

# Iterable Unpacking: example

- Example 1

```
def distance_square(p1, p2):  
    (x1, y1), (x2, y2) = p1, p2  
    dis = (x1 - x2) ** 2 + (y1 - y2) ** 2  
    print(dis)  
  
distance_square((1, 2), (3, 4))
```

- Example 2

```
points = [(1, 2), (2, 2), (3, 4)]  
for p in points:  
    # p 是 tuple  
    print(p[0], p[1])  
  
# 結合 Unpacking  
for x, y in points:  
    # x 和 y 都是 int  
    print(x, y)
```



# Iterable Unpacking: \*args

"\*" can be used to unpack Iterable objects (tuples, lists, etc.)

```
p1 = (1, 2)
p2 = (3, 4)

(x0, y0), (x1, y1) = p1, p2
x0, y0, (x1, y1) = *p1, p2|
(x0, y0), x1, y1 = p1, *p2
x0, y0, x1, y1 = *p1, *p2
```

# Iterable Unpacking: example

- Example 3

```
def norm_square(x, y):  
    return x ** 2 + y ** 2  
  
vector = (3, 4)  
  
# method 1  
print(norm_square(x=vector[0], y=vector[1]))  
  
# method 2  
x, y = vector  
print(norm_square(x, y))  
  
# method 3  
print(norm_square(*vector))
```

- “\*”: unpacking

# Iterable Unpacking: example

- In both packing and unpacking, "\*" cannot be used alone, unless it is used with "," or used as a parameter of a function.

- \* as unpacking

```
p1 = (1, 2)
x0, x1 = *p1
print(x0)
print(x1)
```

File "<ipython-input-51-51bf56d6ba74>", line 2

```
x0, x1 = *p1
          ^
```

SyntaxError: can't use starred expression here

```
p1 = (1, 2)
x0, x1 = *p1,
print(x0)
print(x1)
```

```
1
2
```

- \* as packing

```
x1=1
x2=2
*xx = x1, x2
print(xx)
```

File "<ipython-input-57-691609b11006>", line 3

```
*xx = x1, x2
          ^
```

SyntaxError: starred assignment target must be in a list or tuple

```
x1=1
x2=2
*xx, | = x1, x2
print(xx)
```

```
[1, 2]
```

# Dictionary Unpacking

```
# BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True)
bn_args = {
    'momentum': None,
    'track_running_stats': False,
}
bn1 = BatchNorm2d(2, **bn_args)
bn2 = BatchNorm2d(25, **bn_args)
```

# Outline

- **Environment**
- **Built-in Functions**
- **Packing and Unpacking**
- **Class**

# Class

- **inheritance from nn.Module**

```
class MyNN(nn.Module):  
    def __init__(self, input_size, hidden_size):  
        super().__init__()  
        self.layer = nn.Linear(input_size, hidden_size)  
  
    def forward(self, x):  
        return self.layer(x)
```

# Class: example

```
class ReplayMemory:
    def __init__(self, capacity):
        self._buffer = deque(maxlen=capacity)

    def __len__(self):
        return len(self._buffer)
        __len__ ← len(buffer)

    def __getitem__(self, i):
        return self._buffer[i]
        __getitem__ ← buffer[i]

    def append(self, *transition):
        # (state, action, reward, next_state, done)
        self._buffer.append(tuple(map(tuple, transition)))

    def sample(self, batch_size=1):
        return random.sample(self._buffer, batch_size)
```

```
buffer = ReplayMemory(5000)
buffer.append(state, action, reward, next_state, done)
transitions = memory.sample(20)
```

# Practices



# Practice 1

How to generate io\_channel ?

```
channels = [32, 64, 128, 256, 512]

io_channel = [(32, 64), (64, 128), (128, 256), (256, 512)]

for in_channel, out_channel in io_channel:
    do_something(in_channel, out_channel)
```

# Practice 1

Given

```
channels = [32, 64, 128, 256, 512]
```

How to generate `io_channel` without hardcoding the values?

```
io_channel = [(32, 64), (64, 128), (128, 256), (256, 512)]
```


- Hint: `zip`, `slice`

## Practice 1 : Answer

```
channels = [32, 64, 128, 256, 512]
print(list(zip(channels[:-1], channels[1:])))
io_channel = [(32, 64), (64, 128), (128, 256), (256, 512)]
```

## Practice 1 : Review

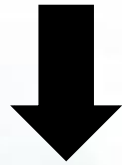
`channels[:-1]`



`channels[1:]`



```
r = zip([32, 64, 128, 256], [64, 128, 256, 512])  
print(list(r))  
# [(32, 64), (64, 128), (128, 256), (256, 512)]
```



```
channels = [32, 64, 128, 256, 512]  
r = zip(channels[:-1], channels[1:])  
print(list(r))  
# [(32, 64), (64, 128), (128, 256), (256, 512)]
```

## Practice 1 : Example

```
channels = [32, 64, 128, 256, 512]

for in_channel, out_channel in zip(channels[:-1], channels[1:])
    do_something(in_channel, out_channel)
```

## Practice 2: Transpose List of List

- Is it possible to transpose a matrix using only one row of code?
- Hint: zip, list, map

```
before = [  
    [ 1,  2,  3,  4,  5],  
    [ 6,  7,  8,  9, 10],  
    [11, 12, 13, 14, 15],  
]
```

```
after = [  
    [1,  6, 11],  
    [2,  7, 12],  
    [3,  8, 13],  
    [4,  9, 14],  
    [5, 10, 15],  
]
```

## Practice 2: Answer

```
before = [  
    [ 1,  2,  3,  4,  5],  
    [ 6,  7,  8,  9, 10],  
    [11, 12, 13, 14, 15],  
]  
  
print(list(map(list, zip(*before))))  
  
after = [  
    [1,  6, 11],  
    [2,  7, 12],  
    [3,  8, 13],  
    [4,  9, 14],  
    [5, 10, 15],  
]
```

## Practice 2: Review

```
r1 = zip([1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15])
print(list(r1))
# [(1, 6, 11), (2, 7, 12), (3, 8, 13), (4, 9, 14), (5, 10, 15)]
```

```
r2 = list(map(list, r1))
# [[1, 6, 11], [2, 7, 12], [3, 8, 13], [4, 9, 14], [5, 10, 15]]
```

```
r1 = zip(*before)
r2 = list(map(list, r1))
```

```
# in one line
result = list(map(list, zip(*before)))
```