

Khoa Toán, Đại học Khoa học Huế

LẬP TRÌNH PYTHON

Python Programming

TS. Nguyễn Hoàng Sơn

Nội dung môn học

- ➊ Tổng quan về Python
- ➋ Ngữ nghĩa, cú pháp LT trong Python
- ➌ Các kiểu dữ liệu phức tạp của Python
- ➍ Lập trình hướng đối tượng trong Python
- ➎ Một số thư viện chuẩn của Python
- ➏ Tài liệu tham khảo

Nội dung môn học

- ❶ Tổng quan về Python
- ❷ Ngữ nghĩa, cú pháp LT trong Python
- ❸ Các kiểu dữ liệu phức tạp của Python
- ❹ Lập trình hướng đối tượng trong Python
- ❺ Một số thư viện chuẩn của Python
- ❻ Tài liệu tham khảo

Giới thiệu Python: Mô tả, đặc trưng

- **Python** là NNLT thông dịch (interpreted), bậc cao và thuộc phổ biến nhất hiện nay, thường được lựa chọn cho nhiều dự án phát triển.
- **Một số đặc trưng nổi bật:**
 - ① **Cú pháp đơn giản:** dễ đọc, dễ viết $\Rightarrow$ tạo ra mã nguồn (source code) dễ bảo trì, phát triển.
 - ② **Đa năng:** có *Thư viện* và *Framework* đa dạng, phong phú $\Rightarrow$ hỗ trợ cho nhiều lĩnh vực: *Web, KHDL, TTNT, tự động hóa và trò chơi*.
 - ③ **Tích hợp tốt:** có khả năng tương tác và tích hợp dễ dàng với các NN khác.
 - ④ **Hỗ trợ cho nhiều nền tảng:** *Windows, Linux ...*
 - ⑤ **Cộng đồng mạnh mẽ:** có 1 cộng đồng lớn và nhiệt tình, cung cấp các TV, công cụ và tài liệu phong phú.
 - ⑥ **Miễn phí và MN mở.**

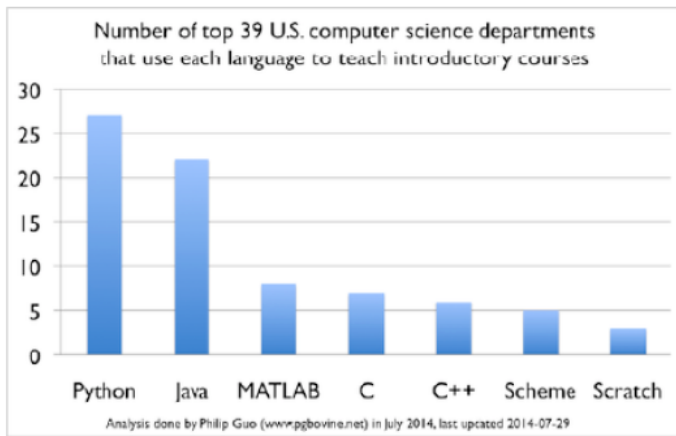
Giới thiệu Python: Xếp hạng các NNLT

- Theo Stack Overflow năm 2021:



Giới thiệu Python: Sử dụng Python

- DLTK về việc sử dụng Python để giảng dạy:



Giới thiệu Python: Lịch sử phát triển

- ① Hình thành vào cuối những năm 1980, và chính thức vào 1989 bởi *Guido Van Rossum* tại **Centrum Wiskunde & Informatica (CWI)** ở Hà Lan.
- ② Năm 2000: Python 2.0
- ③ Năm 2008: Python 3.0
- ④ Hiện nay: Python 3.14.7.

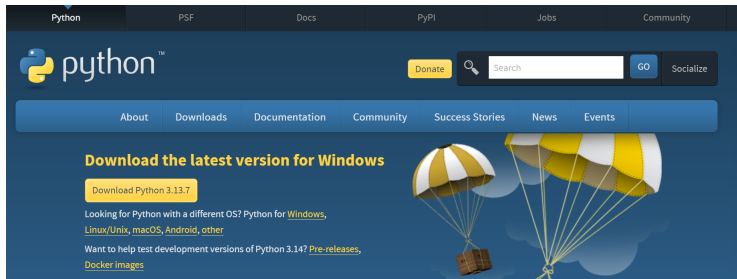


Guido Van Rossum

Cài đặt Python và các trình soạn thảo

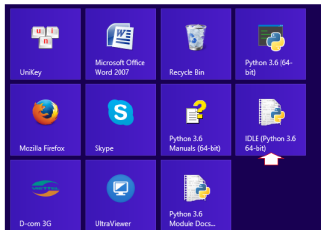
- **Tải và cài đặt Python:**

<https://www.python.org/downloads/>



Cài đặt Python và các trình soạn thảo

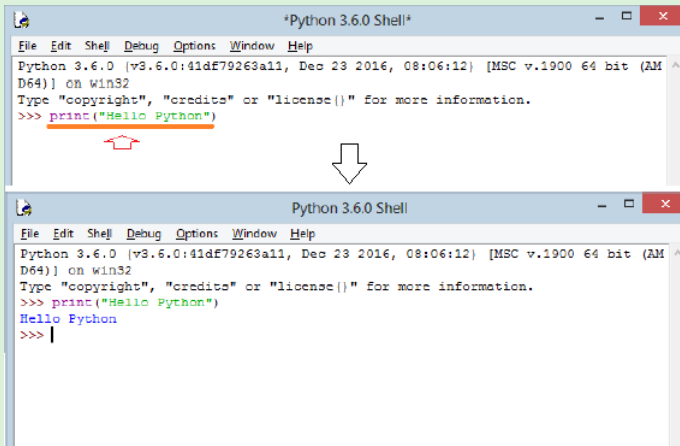
- **Bắt đầu với Python:**
 - Trên Start Menu của Windows, chạy **IDLE** (64bit):



- **IDLE** (Integrated Development and Learning Environment): *môi trường học tập và phát triển tích hợp* đi kèm với cài đặt mặc định của Python. **IDLE** cung cấp 1 giao diện dòng lệnh tương tác đơn giản (**Python Shell**) để soạn thảo, chạy mã nguồn Python và thực hiện một số thao tác cơ bản, thích hợp cho những người mới bắt đầu học LT Python.

Cài đặt Python và các trình soạn thảo

Ví dụ



```
*Python 3.6.0 Shell*
File Edit Shell Debug Options Window Help
Python 3.6.0 (v3.6.0:41df79263a11, Dec 23 2016, 08:06:12) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("Hello Python")
Hello Python
>>> |
```

Cài đặt Python và các trình soạn thảo

- **IDE** (Integrated Development Environment): phần mềm được sử dụng để viết, thử nghiệm và sửa lỗi mã nguồn trong Python.
- Một số IDE khác trong Python ngoài **IDLE** *:
 - ① **Spyder, Jupyter Notebook** (Anaconda): thích hợp cho KHDH, HM
 - ② **PyCharm**: rất mạnh, thích hợp cho KHDH
 - ③ **Visual Studio Code (VSCode)**: thích hợp cho KHDH, HM
 - ④ **Kite**
 - ⑤ **Sublime Text 3**
 - ⑥ **Atom**
 - ⑦ **PyDev**
 - ⑧ **Thonny**
 - ⑨ **ActivePython**

Làm quen với LT Python: Tên

- Là các từ được dùng để đặt cho: *biến, hàm, lớp, mô-đun* và *các ĐT khác*.
- **Quy tắc đặt tên:**
 - ① Phải bắt đầu bằng 1 chữ cái (a-z, A-Z) hoặc dấu gạch dưới (`_`), sau đó có thể chứa các chữ cái, số (0-9), (`_`).
 - ② Không chứa các ký tự trống và đặc biệt như “@, !, \$, %”.
 - ③ Phân biệt chữ hoa và chữ thường.
 - ④ Không được sử dụng các từ khóa của Python để đặt tên biến: “**if, else, while, sum, list ...**”
 - ⑤ Tên nên được đặt sao cho dễ hiểu, đúng với mục đích, chức năng của chúng $\Rightarrow$ mã nguồn dễ đọc, dễ hiểu hơn.

Ví dụ

- ① **Tên hợp lệ:** `'so_chan', 'Nho_hon_0', '_Duong'`
- ② **Tên không hợp lệ:** `'2023data', 'thongke%', 'dai hoc'`

Làm quen với LT Python: Từ khoá

- Từ có ý nghĩa đặc biệt được dành riêng cho NNLT $\Rightarrow$ có vai trò rất quan trọng trong cú pháp và cấu trúc của 1 CT Python.
- Không được dùng từ khóa đặt tên cho *biến*, *hàm*, *lớp* hoặc *bất kỳ đối tượng nào* trong CT của mình.
- **Danh sách các từ khóa chính trong Python:**

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Làm quen với LT Python: Từ đặc biệt

- Từ có ý nghĩa đặc biệt trong ngữ cảnh cụ thể, giúp Python trở thành 1 NN linh hoạt và mạnh, cho phép chúng ta tùy chỉnh hành vi của ĐT và tích hợp dễ dàng với các cấu trúc NN có sẵn, đặc biệt là khi làm việc với lớp hoặc mô-đun:

self	cls	class_
------	-----	--------

Làm quen với LT Python: Hàm

- Một khối mã có thể được gọi để thực hiện 1 tác vụ cụ thể $\Rightarrow$ giúp tái sử dụng mã, tạo CT để dễ đọc hơn và tách biệt các phần khác nhau của CT.
- **Một số hàm thông dụng trong Python:**

print	input	len	type	int
float	str	list	dict	range
sum	max	min	sorted	zip
map	filter	enumerate	abs	round
open	close	str.format	join	split

Làm quen với LT Python: Lệnh, khối lệnh

- Các đoạn mã thực hiện 1 công việc cụ thể nào đó: các dòng mã đơn giản hoặc các khối mã phức tạp.
- **Cách viết lệnh:**
 - ① **Lệnh** (ngắn): được viết trên 1 dòng và ký tự xuống dòng nghĩa là kết thúc lệnh đó.
 - ② **Lệnh dài** (muốn viết trên nhiều dòng): sử dụng dấu “\” cuối câu lệnh để chỉ rằng lệnh bao gồm cả dòng tiếp theo.
 - ③ **Nhiều câu lệnh** (trên 1 dòng): sử dụng dấu chấm phẩy “;” để ngăn cách giữa các câu lệnh.
- **Khối lệnh:** là các lệnh liên tiếp có cùng khoảng cách thụt đầu dòng.

Làm quen với LT Python: Lệnh, khối lệnh- Ví dụ

Ví dụ

① Lệnh trên nhiều dòng:

```
value = 1 + \  
2 + \  
3
```

② Nhiều lệnh trên 1 dòng:

```
a = 'One'; b = "Two"; c = "Three"
```

③ Khối lệnh:

```
if True:
```

```
    print("Hello")
```

```
    print("True")
```

```
else:
```

```
    print("False")
```

Làm quen với LT Python: Chuỗi

- Một 1 loại DL được dùng để lưu trữ và làm việc với văn bản, chúng có thể chứa bất kỳ ký tự nào (*chữ cái, chữ số, ký tự đặc biệt, khoảng trắng*).
- **Cách biểu diễn chuỗi:**
 - ① **Chuỗi trên 1 dòng:**
 - Cặp dấu nháy đơn (' ')
 - Cặp dấu nháy kép (" ").
 - ② **Chuỗi trên nhiều dòng:**
 - Cặp 3 dấu nháy kép (" " " ").

Ví dụ

① Chuỗi trên 1 dòng:

```
str1 = 'Hello every body'
str2 = "Hello Python"
```

② Chuỗi trên nhiều dòng:

```
multiLineStr = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

Làm quen với LT Python: Chú thích

- Giải thích mã nguồn và thêm ghi chú cho mã của bạn. Khi thông dịch, Python sẽ bỏ qua những chú thích này.
- **Một số cách để viết chú thích:**
 - ① **Chú thích dòng đơn** (1 dòng): bắt đầu bằng kí tự thăng (#), sau đó mọi thứ sau (#) được xem chú thích.
 - ② **Chú thích nhiều dòng:**
 - dùng nhiều chú thích dòng đơn.
 - cặp 3 dấu nháy đơn ''' '''
 - cặp 3 dấu nháy kép """ """

Ví dụ

```
# This is a comment  
print("Python Programming")
```

Làm quen với LT Python: Biến

- Là 1 ĐT mà giá trị của nó có thể thay đổi được khi thực hiện CT. Mỗi biến đều được đặt tên và cần 1 lượng ô nhớ MT tương ứng để lưu giá trị của nó.
- Trong Python, mỗi biến là 1 con trỏ chỉ đến ô nhớ chứa giá trị đã được gán cho biến đó.
- **Khai báo biến:**
 - Chỉ cần gán 1 giá trị cho tên biến bằng dấu “bằng” (=):

$\langle TB \rangle = \langle \text{giá trị} \rangle$

- Python tự động xác định KDL của biến dựa vào giá trị được gán.
 - Python không cần lệnh khai báo biến.
- **Sử dụng biến:** Python dùng biến để thực hiện các phép toán và tương tác với DL.

Làm quen với LT Python: Biến-Lưu ý

- ① **Các ô nhớ trong MT:** được đánh địa chỉ (số thứ tự) bắt đầu từ 0.
- ② **Hàm lấy địa chỉ ô nhớ lưu trữ giá trị của biến:**

id(< TB >)

- ③ Các ô nhớ được sử dụng để chứa giá trị của biến. Trong quá trình thay đổi giá trị, nếu ô nhớ nào không có biến chỉ tới thì ô nhớ đó sẽ được giải phóng để chứa các giá trị khác trong quá trình thực hiện CT.
- $\leadsto$ Cùng 1 CT, mỗi lần chạy địa chỉ ô nhớ của biến có thể khác nhau.

Làm quen với LT Python: Biến-Ví dụ

Ví dụ.

```
x = 10 # int (số nguyên)
name = "Alice" # str (chuỗi)
pi = 3.14159 # float (số thực)
is_student = True # bool (boolean)
print("Địa chỉ ô nhớ: ", id(x))
```

Kết quả:

Địa chỉ ô nhớ: 140728222754896

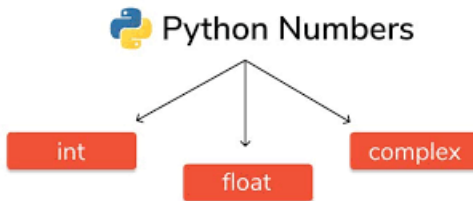
Làm quen với LT Python: Biến-Viết gọn

- Một số **cách viết ngắn gọn** các biểu thức gán giá trị trong Python:

Cách gán thông thường	Cách gán rút gọn
$a = 1$ $b = 5$ $c = 10$	$a, b, c = 1, 5, 10$
$x = a$ $a = b$ $b = x$	$a, b = b, a$
$a = 10$ $b = 10$ $c = 10$	$a = b = c = 10$

Làm quen với LT Python: Các KDL cơ bản

- Python hỗ trợ 3 KDL cơ bản để biểu diễn các số:
 - 1 Nguyên (int)
 - 2 Thực (float)
 - 3 Phức (complex).



- Xem **KDL** của biến:

type(< TB >)

Làm quen với LT Python: Kiểu số nguyên-(1)

- Các phép toán:

Phép toán	Ký hiệu	Ví dụ
Cộng	+	$1 + 2 = 3$
Trừ	-	$5 - 1 = 4$
Nhân	*	$2 * 5 = 10$
Lũy thừa	**	$3 ** 2 = 9$
Chia	/	$4 / 2 = 2$
Chia lấy thương	//	$5 // 2 = 2$
Chia lấy dư	%	$5 \% 2 = 1$

- ↪ Python cho phép tính toán với số nguyên có giá trị rất lớn.

Làm quen với LT Python: Kiểu số nguyên-(2)

Ví dụ.

```
n=2021
k=100
T=n**k
print("gia tri T la: ",T)
```

Kết quả:

gia tri T la:

```
1898085081608942676501785937485816750085126121649100
1491235655161857712925195294212825926647506000551692
8951858312990367112874013108609073422620559795539416
5075391001
```

Làm quen với LT Python: Kiểu số thực

- Các phép toán:

Phép toán	Ký hiệu	Ví dụ
Cộng	+	$1.0 + 2.0 = 3.0$
Trừ	-	$5.0 - 1.0 = 4.0$
Nhân	*	$2.0 * 5.0 = 10.0$
Lũy thừa	**	$25 * 0.5 = 5.0$
Chia	/	$3/2 = 1.5$

- $\leadsto$ Số thực có thể viết dưới dạng ký pháp khoa học:
 - $\hookrightarrow$ Số $2.0 * 10^3 \Rightarrow$ viết lại 2.0e3.

Làm quen với LT Python: Kiểu số phức-(1)

- **Số phức:** $a + bj$
 - $\leadsto$
 - a, b : số thực
 - a : phần thực
 - b : phần ảo.
- **Hàm tạo số phức:**
 - **`complex`**(a, b) = $a + bj$.
 - $b = 0 \Rightarrow$ viết gọn **`complex`**(a).

Ví dụ.

```
x = 3 + 4j
y = complex(1,5)
print("x + y = ", x + y)
```

Kết quả:

```
x + y = (4 + 9j)
```

Làm quen với LT Python: Kiểu số phức-(2)

- Một số PT:
 - ① Truy cập phần thực: **.real**
 - ② Truy cập phần ảo: **.imag**
 - ③ Tính giá trị tuyệt đối: **abs()**
 - ④ Tính số phức liên hợp: **.conjugate()**

Ví dụ.

- $x = 3 + 4i$
- $\Rightarrow$
 - $3 = x.\text{real}$
 - $4 = x.\text{imag}$
 - $\text{abs}(x) = 5$
 - $3 - 4i = x.\text{conjugate}()$

Làm quen với LT Python: Hàm tạo số NN

- Một số hàm cơ bản (TV **random**):
 - ① **random()**: tạo 1 số thực NN (PP đều) thuộc $[0.0, 1.0)$.
 - ② **uniform**(a, b): tạo 1 số thực NN thuộc $[a, b)$.
 - ③ **randint**(a, b): tạo 1 số nguyên NN thuộc $[a, b]$.
 - ④ **randrange**($start, stop, step$): tạo 1 số nguyên NN thuộc $[start : stop - 1]$ với bước nhảy $step$.
 - ① **randrange**($start, stop, 1$) = **randrange**($start, stop$)
 - ② **randrange**($0, stop, step$) = **randrange**($stop, step$)
 - ⑤ **choice**($sequence$): chọn NN 1 phần tử từ dãy (chuỗi, DS, bộ).
 - ⑥ **shuffle**($sequence$): xáo trộn NN các phần tử trong dãy (DS, bộ).
- $\leadsto$ Tạo DL thử nghiệm, mô phỏng.

Làm quen với LT Python: Hàm toán học-(1)

- Một số hàm toán học thông dụng:

Tên hàm	Thực hiện
$\sin(x)$	Tính sin của 1 góc có số đo x (rad)
$\cos(x)$	Tính cosin của 1 góc có số đo x (rad)
$\tan(x)$	Tính tang của 1 góc có số đo x (rad)
$\sqrt{x}$	Tính căn bậc 2 của x
$\text{trunc}(x)$	Tính phần nguyên của x
π	Tính giá trị của số pi
$\text{round}(x, n)$	Làm tròn x đến n chữ số thập phân
$\text{abs}(x)$	Tính giá trị tuyệt đối x
$\max(x_1, \dots, x_n)$	Trả về GTLN
$\min(x_1, \dots, x_n)$	Trả về GTNN

Làm quen với LT Python: Hàm toán học-(2)

- Nạp mô-đun **math** để sử dụng được các hàm:

from math *import* *

- $\leadsto$ Ngoại trừ các hàm cơ bản: **abs()**, **min()**, **max()**, **round()**.
- Liệt kê tất cả hàm toán học:

dir(math)

- Tìm hiểu ý nghĩa 1 hàm toán học:

help(math.<tên hàm >)

- $\leadsto$ Khi tính toán với các số thực, MT có thể dẫn đến sai lệch về giá trị $\Rightarrow$ nên sử dụng hàm **round()**.

Làm quen với LT Python: Rút gọn phép toán

- Một số cách viết phép toán rút gọn:

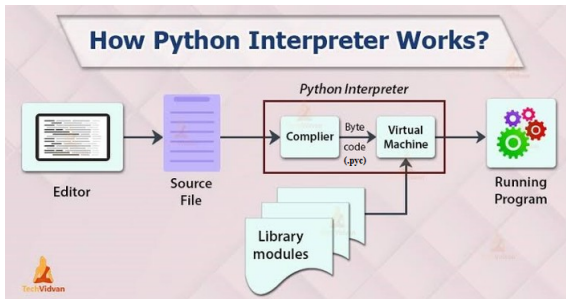
Biểu thức	Rút gọn
$n = n \pm a$	$n \pm = a$
$n = n * a$	$n * = a$
$n = n / a$	$n / = a$
$n = n // a$	$n // = a$
$n = n ** a$	$n ** = a$

- $\leadsto$ Mã nguồn ngắn, gọn và dễ đọc hơn.

Nội dung môn học

- ① Tổng quan về Python
- ② Ngữ nghĩa, cú pháp LT trong Python
- ③ Các kiểu dữ liệu phức tạp của Python
- ④ Lập trình hướng đối tượng trong Python
- ⑤ Một số thư viện chuẩn của Python
- ⑥ Tài liệu tham khảo

Chương 2. Ngữ nghĩa và cú pháp LT trong Python



- **Cú pháp** (Syntax): quy tắc, quy định về cách mã (code) phải viết để nó có thể được hiểu và thực thi.
- **Ngữ nghĩa** (Semantics): ý nghĩa của mã sau khi nó đã được viết đúng cú pháp. Nó xác định cách mã hoạt động và tương tác với DL, biến trong CT.

Chương 2. Ngữ nghĩa và cú pháp LT ...

- 1 Xuất/Nhập DL
- 2 Các cấu trúc điều khiển
- 3 Xử lý ngoại lệ
- 4 Hàm
- 5 Mô-đun và gói



Xuất/Nhập dữ liệu: Xuất DL ra màn hình

- **Cú pháp:**

- ① **`print`**(*value*₁, *value*₂, ..., ***sep*** = "...", ***end*** = "...")
 - *value*₁, *value*₂, ...: các giá trị hoặc biểu thức muốn xuất ra màn hình.
 - ***sep***: ký tự ngăn cách giữa các giá trị, mặc định là dấu cách.
 - ***end***: ký tự kết thúc của lệnh **`print`**, mặc định là dòng mới.
- ② **`print`**(*f-string*)
 - Xuất chuỗi *string* ra màn hình và thực hiện tính toán giá trị các biến nếu chúng được nhúng trong chuỗi qua {}.
- ③ **`print`**(string.***format***())
 - Xuất chuỗi *string* ra màn hình với các vị trí cụ thể để chèn các giá trị vào chuỗi qua PT.***format***()).

Xuất/Nhập dữ liệu: Xuất DL: Ví dụ-(1)

Ví dụ.

```
print("a", "b", "c")  
print("a", "b", "c", sep = "&")  
  
print("x", end = "+")  
print("y")  
print("Lập trình Python")
```

Kết quả:

a b c

a&b&c

x+y

Lập trình Python

Xuất/Nhập dữ liệu: Xuất DL: Ví dụ-(2)

Ví dụ.

```
a = 2
```

```
b = 10
```

```
c = a + b
```

```
print(f"1. Tổng của {a} và {b} là {c}")
```

```
print("2. Tổng của {} và {} là {}".format(a, b, c))
```

Kết quả:

1. Tổng của 2 và 10 là 12

2. Tổng của 2 và 10 là 12

Xuất/Nhập dữ liệu: Nhập DL từ bàn phím

- **Cú pháp:**

$\langle TB \rangle = \text{input}()$

- Nhập DL vào từ bàn phím và lưu trữ giá trị được nhập vào biến.
- $\leadsto$ Hàm ***input***: luôn trả về 1 chuỗi ký tự $\Rightarrow$ nếu muốn kết quả trả về là các số $\Rightarrow$ chuyển đổi kiểu DL (type conversion - ***ép kiểu***):
 - ① **int**("10") $\Rightarrow$ "10" được chuyển thành số nguyên 10
 - ② **float**("10") $\Rightarrow$ "10" được chuyển thành số thực 10.0
 - ③ **complex**("10") $\Rightarrow$ "10" được chuyển thành số phức $10 + 0j$

Xuất/Nhập dữ liệu: Nhập DL: Ví dụ

Ví dụ.

```
x = input("Nhập vào DL:")  
print("DL được nhập vào: ", x)  
  
a = int(input("Nhập vào số a = "))  
b = int(input("Nhập vào số b = "))  
print("Tổng a + b = ", a + b)
```

Xuất/Nhập dữ liệu: Nhập DL: Lưu ý

- 1 Các hàm chuyển đổi kiểu:

int(), *float()*, *complex()*, *str()*, *bool()*

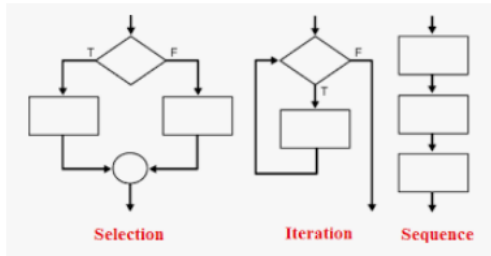
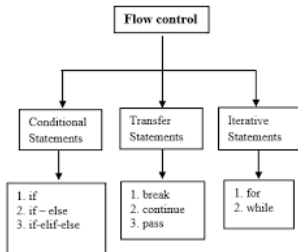
- 2 Hàm kiểm tra kiểu các biến:

type()

- 3 Chuyển đổi kiểu DL có thể gây ra mất mát thông tin: số *thực* chuyển thành số *nguyên* $\Rightarrow$ cắt bỏ phần thập phân.

Các cấu trúc điều khiển

- **Cấu trúc điều khiển:** cách chúng ta điều khiển luồng thực thi của CT dựa trên các điều kiện và sự lặp đi lặp lại: *if, for, while*



Các cấu trúc điều khiển: Cấu trúc rẽ nhánh - (1)

- ① ***if*** <điều kiện>:
 < khối lệnh >
 - ② ***if*** <điều kiện>:
 <khối lệnh của *if*>
 else :
 <khối lệnh của *else*>
 - ③ ***if*** <điều kiện 1>:
 <khối lệnh của *if*>
 elif <điều kiện 2>:
 <khối lệnh của *elif*>
 else :
 <khối lệnh của *else* >
- Thực hiện các khối lệnh của các điều kiện tương ứng.

Các cấu trúc điều khiển: Cấu trúc rẽ nhánh - (2)

- ~→
 - ① ***elif*** ($\equiv$ ***else if***): có thể không có hoặc có nhiều.
 - ② ***if...elif...elif...***: sự thay thế cho câu lệnh ***switch*** hay ***case*** trong các NNLT khác.
 - ③ Các lệnh ***if...*** có thể lồng nhau.
- **Cách viết gọn** của một số biểu thức điều kiện:

Cách viết đầy đủ	Cách viết gọn
<i>if</i> <điều kiện> == <i>True</i> :	<i>if</i> <điều kiện>:
<i>if</i> <i>a</i> == 0 <i>and</i> <i>b</i> == 0:	<i>if</i> <i>a</i> == <i>b</i> == 0:
<i>if</i> 1 < <i>a</i> <i>and</i> <i>a</i> < <i>b</i> <i>and</i> <i>b</i> < 3:	<i>if</i> 1 < <i>a</i> < <i>b</i> < 3:

Các cấu trúc điều khiển: Cấu trúc rẽ nhánh - (3)

- Các phép toán so sánh trong các điều kiện:

Phép toán	Ký hiệu
Bằng	<code>==</code>
Khác	<code>!=</code>
Nhỏ hơn	<code><</code>
Lớn hơn	<code>></code>
Nhỏ hơn hoặc bằng	<code><=</code>
Lớn hơn hoặc bằng	<code>>=</code>

- Kết quả của phép so sánh: *True* hoặc *False*
- Python có hỗ trợ các phép toán logic: **not**, **and**, **or**
- Thứ tự ưu tiên: thấp hơn các phép toán số học, so sánh.

Các cấu trúc điều khiển: Cấu trúc lặp

① Lặp với số lần lặp xác định trước:

for < TB > ***in*** <đối tượng>:

< khối lệnh >

- Biến lần lượt nhận giá trị trong đối tượng (*chuỗi*, *DS*, *bộ*, *TĐ*). Mỗi lần biến nhận 1 giá trị thì khối lệnh thực hiện 1 lần.

② Lặp với số lần lặp chưa xác định trước:

while <điều kiện>:

<khối lệnh>

- Nếu điều kiện đúng $\Rightarrow$ thực hiện khối lệnh.

Các cấu trúc điều khiển: Cấu trúc lặp-Ví dụ

Ví dụ.

```
for x in [1,2]:  
    print(x)
```

Kết quả:

1
2

Ví dụ.

```
n = 1  
while (n < 3):  
    print("n = ", n)  
    n += 1
```

Kết quả:

n = 1
n = 2

Các cấu trúc điều khiển: pass, continue, break

- ① **pass:** lệnh không làm gì cả. Nó thường được sử dụng khi bạn cần 1 câu lệnh để hoàn thành cú pháp, chẳng hạn khi chúng ta định nghĩa 1 hàm hoặc 1 lớp mà chưa muốn triển khai nội dung cho nó.
 - ② **continue:** được sử dụng trong vòng lặp (**for, while**) để bỏ qua phần còn lại của vòng lặp và chuyển đến vòng lặp tiếp theo. Thường được sử dụng khi muốn bỏ qua một số phần tử hoặc điều kiện cụ thể trong vòng lặp và tiếp tục với lần lặp kế tiếp.
 - ③ **break:** được sử dụng trong vòng lặp (**for, while**) để thoát khỏi vòng lặp ngay lập tức khi 1 điều kiện đúng được đáp ứng. Thường được sử dụng nếu muốn kết thúc vòng lặp sớm khi 1 điều kiện cụ thể xuất hiện.
- ~> Đây là các lệnh đặc biệt được sử dụng trong cấu trúc điều khiển để kiểm soát luồng thực thi của CT.

Các cấu trúc điều khiển: ví dụ - (1)

Ví dụ. (pass)

```
a = 2  
b = 10  
if b > a:  
    pass
```

Ví dụ. (break)

```
for x in [1,2,3,4,5]:  
    if(x == 3):  
        break  
    print("x = ", x)
```

Kết quả:

```
x = 1  
x = 2
```

Các cấu trúc điều khiển: ví dụ - (2)

Ví dụ. (continue)

```
for x in [1,2,3,4,5]:  
    if (x % 2 == 0):  
        continue  
    print(f"x = {x} là số chẵn")  
    print(f"x = {x} là số lẻ")
```

Kết quả:

```
x = 1 là số lẻ  
x = 3 là số lẻ  
x = 5 là số lẻ
```

Xử lý ngoại lệ

- **Ngoại lệ** (exception): sự kiện xảy ra trong quá trình thực thi CT $\Rightarrow$ việc thực thi bị gián đoạn (*lỗi hoặc tình huống không mong muốn*):
 - ① Chia cho số 0.
 - ② Mở tệp không tồn tại.
 - ③ Sử dụng 1 biến chưa được định nghĩa ...
- $\Rightarrow$ CT không nhận được kết quả như mong đợi hoặc rối loạn hệ thống.
- $\Rightarrow$ **Xử lý ngoại lệ** (exception handling) $\Rightarrow$ CT vẫn có thể thực thi 1 cách an toàn cũng như kiểm soát được luồng thực thi.
- $\leadsto$ **Xử lý ngoại lệ**: phần quan trọng trong LT.



Xử lý ngoại lệ: try ... except

- **Cú pháp lệnh:**

try :

<khối lệnh 1>

except :

<khối lệnh 2>

- Thực hiện khối lệnh 1. Trong quá trình thực hiện khối lệnh 1 nếu có phát sinh lỗi $\Rightarrow$ khối lệnh 2 sẽ thực hiện.
- $\leadsto$
 - ① **try:** bao bọc mã muốn kiểm tra lỗi.
 - ② **except:** xử lý ngoại lệ.



Xử lý ngoại lệ: try ... except-Ví dụ

Ví dụ. (ZeroDivisionError)

```
a = int(input("Nhập vào số a:"))
b = int(input("Nhập vào số b:"))
try:
    c = a/b
    print(f"Thương của a và b là {round(c, 3)}")
except:
    print("Không thể chia 0!")
print("CT kết thúc")
```

- ~> Có thể bắt nhiều loại ngoại lệ trong cùng 1 khối **try**
... **except**

Xử lý ngoại lệ: `try ... except ... else`

- **Cú pháp lệnh:**

try :

<khối lệnh 1>

except :

<khối lệnh 2>

else : # tùy chọn

<khối lệnh 3>

- Thực hiện khối lệnh 1. Nếu trong quá trình thực hiện khối lệnh 1 có phát sinh lỗi $\Rightarrow$ khối lệnh 2 sẽ được thực hiện. Nếu khối lệnh 1 không phát sinh lỗi $\Rightarrow$ khối lệnh 3 sẽ thực hiện.

Xử lý ngoại lệ: try ... except ... else-Ví dụ

Ví dụ. (ZeroDivisionError-else)

```
a = int(input("Nhập vào số a: "))
b = int(input("Nhập vào số b: "))
try:
    c = a/b
except:
    print("Lỗi chia 0!")
else:
    print(f"Thương của a và b là {round(c, 3)}")
print("CT kết thúc.")
```

Xử lý ngoại lệ: `try ... except ... else ... finally`

- **Cú pháp lệnh:**

try :

<khối lệnh 1>

except :

<khối lệnh 2>

else :

<khối lệnh 3>

finally : # tùy chọn

<khối lệnh 4>

- Sau khi các khối lệnh trong (***try*** và ***else***) hoặc ***except*** đã được thực thi, khối lệnh 4 trong ***finally*** được thực hiện bất kể có ngoại lệ hay không.
- ~> Khối ***finally*** thường được sử dụng để thực hiện *giải phóng tài nguyên, đóng kết nối hoặc dọn dẹp các thao tác cần thiết.*

Xử lý ngoại lệ: try ... except ... else ... finally-Ví dụ

Ví dụ. (ZeroDivisionError-finally)

```
a = int(input("Nhập vào số a: "))
b = int(input("Nhập vào số b: "))
try:
    c = a/b
except:
    print("Không thể chia 0")
else:
    print(f"Thương của a và b là {round(c, 3)}")
finally:
    print("Thực hiện khối lệnh try ... finally")
print("CT kết thúc.")
```

Xử lý ngoại lệ: Exception

- **Exception:** là lớp cơ sở cho tất cả ngoại lệ chuẩn trong Python.
- ~→
 - ① Thường thì dùng **try ... except** để bắt cụ thể 1 ngoại lệ (*ZeroDivisionError, IndexError ...*).
 - ② Khi không biết chính xác loại ngoại lệ nào có thể xảy ra
⇒ có thể bắt tất cả ngoại lệ bằng cách dùng lớp **Exception**:
 - **except Exception as e:**
 - Biến **e**: giúp truy cập thông tin về lỗi đã xảy ra, chẳng hạn như thông báo lỗi, hoặc các thuộc tính khác của đối tượng ngoại lệ
 - ⇒ giúp CT không bị dừng một cách đột ngột.

Xử lý ngoại lệ: Exception-Ví dụ - (1)

Ví dụ. (Exception - lớp lỗi *ZeroDivisionError*)

```
a = 5
```

```
b = 0
```

```
try:
```

```
    c = a/b
```

```
except Exception as e:
```

```
    print(f"Thông báo lỗi: {e} \nLoại lỗi: {type(e)}")
```

Kết quả:

Thông báo lỗi: division by zero

Loại lỗi: <class 'ZeroDivisionError'>

Xử lý ngoại lệ: Exception-Ví dụ - (2)

Ví dụ. (Exception - lớp lỗi *IndexError*)

```
L = [1, 2, 3, 4, 5]
```

```
try:
```

```
    x = L[10]
```

```
except Exception as e:
```

```
    print(f"Đã xảy ra ngoại lệ: {e} \nThuộc lớp: {type(e)}")
```

Kết quả:

Đã xảy ra ngoại lệ: list index out of range

Thuộc lớp: <class 'IndexError'>

Xử lý ngoại lệ: Các lớp Exception - (1)

- Một số lớp Exception thường gặp:

Tên lớp	Lỗi	Ví dụ
<i>ImportError</i>	Import sai tên, PT trong 1 module hay package	try: from sys import ABC except Exception as e: print(type(e),e)
<i>IndexError</i>	Truy cập đến chỉ số không thuộc DS, string, tuple ...	L = [1,2,3,4,5] try: x = L[10] except Exception as e: print(type(e), "-", e)
<i>NameError</i>	Sử dụng biến có tên chưa xác định	try: x = v except Exception as e: print(type(e), "-", e)

Xử lý ngoại lệ: Các lớp Exception - (2)

Tên lớp	Lỗi	Ví dụ
<i>KeyError</i>	Truy cập đến phần tử <u>có khóa không thuộc từ điển</u>	<pre>D = {1:"A", 2:"B"} try: x = D[10] except Exception as e: print(type(e), "-", e)</pre>
<i>TypeError</i>	Tính toán trên các ĐT <u>có kiểu không phù hợp</u>	<pre>try: x = 5/"5" except Exception as e: print(type(e), "-", e)</pre>
<i>ValueError</i>	Tính toán trên các ĐT phù hợp nhưng <u>giá trị không phù hợp</u>	<pre>try: x = int("a") except Exception as e: print(type(e), "-", e)</pre>

Xử lý ngoại lệ: Các lớp Exception - (3)

Tên lớp	Lỗi	Ví dụ
<i>ZeroDivisionError</i>	Chia cho 0	try: $x = 5/0$ except Exception as e: print(type(e),e)
<i>FileNotFoundError</i>	Không tìm thấy tệp để đọc	try: $f = \text{open}("T.txt", "r")$ except Exception as e: print(type(e),e)

Hàm: Khái niệm - (1)

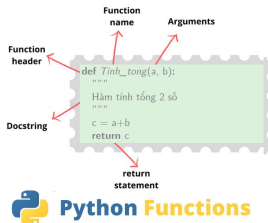
- **Hàm** (function): khối mã độc lập được đặt tên, được thiết kế để thực hiện 1 nhiệm vụ cụ thể hoặc tính toán giá trị.
- ~→
 - ① Tổ chức mã thành các phần nhỏ, dễ quản lý và tái sử dụng.
 - ② Một hàm thường nhận đầu vào (*tham số*), thực hiện các thao tác trên đầu vào và có thể trả về 1 kết quả đầu ra (*giá trị trả về*).
- **Cấu trúc hàm trong Python:**

```
def <tên hàm>([các tham số]):
    [docstrings]
    <khối lệnh>
    return [biểu thức] # tùy chọn
```

Hàm: Khái niệm - (2)



- ① **return**: có thể chứa biểu thức được tính toán và giá trị trả về. Nếu không có biểu thức nào trong **return** hoặc không có **return** trong hàm $\Rightarrow$ hàm sẽ trả về *None*
- ② **docstrings** (tài liệu hàm): chuỗi đặt trong cặp 3 dấu nháy kép (`"""`), mô tả về hàm.



- **Gọi hàm:** `<tên hàm>([các tham số])`
 - $\leadsto$ Các tham số: cách nhau bằng dấu `,` nếu có nhiều (≥ 2).

Hàm: Khái niệm-Ví dụ - (1)

Ví dụ. (Hàm không có tham số)

```
def Tam_giac():  
    """ Hàm (không có tham số) in hình tam giác """  
    print("*")  
    print("**")  
    print("***")  
    print("****")  
    print("*****")  
  
# Gọi hàm  
Tam_giac()
```

Hàm: Khái niệm-Ví dụ - (2)

Ví dụ. (Hàm có tham số)

```
def ting_tong(a, b):
```

```
    """
```

```
    Hàm tính tổng 2 số
```

```
    """
```

```
    c = a+b
```

```
    return c
```

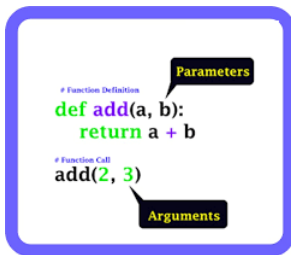
```
a = int(input("Nhập vào 1 số: "))
```

```
b = int(input("Nhập vào 1 số: "))
```

```
print(f"Tổng của {a} và {b}: {ting_tong(a, b)}")
```

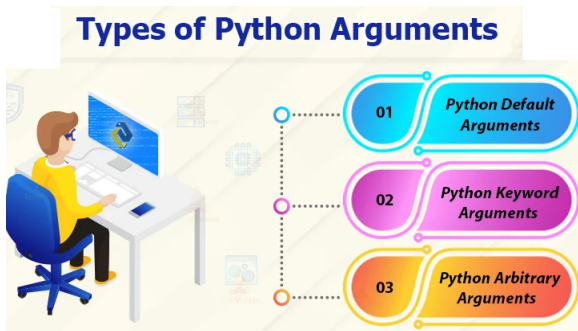
Hàm: Tham số

- **Tham số** (parameter): các biến khai báo trong hàm, được sử dụng để truyền giá trị vào hàm khi gọi nó.
- ~→
 - ① Cho phép truyền thông tin (DL) cần thiết vào hàm $\Rightarrow$ để thực hiện công việc cụ thể trong khối lệnh.
 - ② Có vai trò rất quan trọng trong việc xác định cách hàm hoạt động.



Hàm: Đối số - (1)

- **Đối số** (argument): giá trị được truyền vào hàm khi nó được gọi.
- ~> Ở phía
 - Người gọi: nó là **đối số**
 - Hàm: nó được xem **tham số**.



Hàm: Đối số - (2)

- Các loại đối số trong Python:

- ① **Đối số từ khóa** (keyword):

- Đối số bắt buộc cần phải truyền vào hàm khi gọi nó
 - Hàm sẽ không hoạt động đúng nếu thiếu bất kỳ đối số từ khóa nào.

- ② **Đối số mặc định** (default, tùy chọn):

- Đối số có giá trị mặc định được định nghĩa trong khai báo hàm.
 - Nếu không truyền giá trị cho các đối số này khi gọi hàm $\Rightarrow$ giá trị mặc định sẽ được sử dụng.

- ③ **Đối số tùy ý** (arbitrary): các đối số có thể thay đổi về số lượng cho mỗi lần gọi:

***def** <tên hàm>(* <các tham số tuple>):*

***def** <tên hàm>(** <các tham số dictionary>):*

Hàm: Tham số-Ví dụ - (1)

Ví dụ. (Hàm với tham số mặc định và bắt buộc)

```
def ting_tong(a, b = 0):  
    """Tính tổng của hai số a và b  
    a: tham số bắt buộc, b: tham số mặc định"""  
    tong = a + b  
    return tong  
  
# Gọi hàm với tham số a-bắt buộc, b-mặc định  
print(f"1. Tổng là: {ting_tong(5)}")  
  
# Gọi hàm và truyền cả hai tham số a và b  
print(f"2. Tổng là: {ting_tong(5, 10)}")
```

Hàm: Tham số-Ví dụ - (2)

Ví dụ. (Hàm với tham số, tuple, biến đổi)

```
def  tinh_tong(*t):  
    S = 0  
    for i in t:  
        S += i  
    return S  
print(f"1. Tổng: { tinh_tong()}")  
print(f"2. Tổng: { tinh_tong(5)}")  
print(f"3. Tổng: { tinh_tong(1, 2, 6)}")
```

Hàm: Tham số-Ví dụ - (3)

Ví dụ. (Hàm với tham số, dictionary, biến đổi)

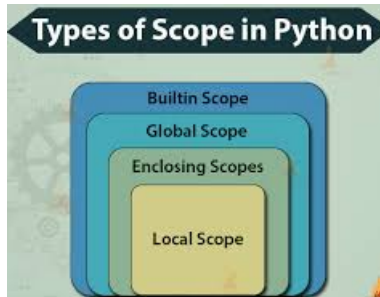
```
def thong_tin(**d): # dictionary
    for key, value in d.items():
        print(f"{key}: {value}")
```

```
# Gọi hàm và truyền các tham số từ điển
thong_tin(Ten = "Tom", Tuoi = 10, Que = "Huế")
```

Hàm: Phạm vi biến-biến cục bộ, toàn cục - (1)

- **Phạm vi của biến:** vị trí, thời gian mà biến có thể truy cập và sử dụng.
- Các loại biến của Python:
 - ① **Biến cục bộ** (local)
 - **Xác định:** định nghĩa bên trong hàm
 - **Phạm vi sử dụng:** chỉ có thể truy cập ở bên trong hàm
 - **Thời gian tồn tại:** chỉ trong thời gian thực thi của hàm.
 - ② **Biến toàn cục** (global)
 - **Xác định:** định nghĩa ở mức cao nhất của CT, ngoài tất cả các hàm
 - **Phạm vi sử dụng:** có thể được truy cập từ bất kỳ đâu trong CT, bao gồm cả trong hàm
 - **Thời gian tồn tại:** trong suốt thời gian CT chạy.

Hàm: Phạm vi biến-biến cục bộ, toàn cục - (2)



- ~> Khi cần sử dụng biến:
 - Hãy xác định phạm vi của biến đó và đảm bảo truy cập nó theo cách đúng
 - Nếu cố gắng truy cập biến cục bộ từ bên ngoài hàm, hoặc không khai báo biến toàn cục trước khi sử dụng nó
⇒ CT có thể gặp lỗi.

Hàm: Biến cục bộ, toàn cục-Ví dụ - (1)

Ví dụ. (Biến toàn cục, cục bộ)

`n = 5` # biến toàn cục

`i = 10` # biến toàn cục

def *tong()*:

`i = 0` # biến cục bộ

`S = 0` # biến cục bộ

while `i <= n`:

`S += i`

`i += 1`

return `S`

print(`f"Tổng các số từ 1 --> {n}: {tong()}"`)

print(`f"Biến i ở ngoài CT con là biến toàn cục: {i}"`)

Hàm: Biến cục bộ, toàn cục-Ví dụ - (2)

Ví dụ. (Lỗi chưa tham chiếu-UnboundLocalError)

```
n = 10
```

```
S = 0 # S là biến toàn cục
```

```
def tong():
```

```
    i = 0
```

```
    while i <= n:
```

```
        S += i # S được hiểu là biến cục bộ trong hàm này,
```

```
        i += 1 # không thay đổi giá trị S trong hàm này được
```

```
    return S
```

```
print(f"Tổng các số từ 1 --> {n}: {tong()}")
```

Hàm: Biến cục bộ, toàn cục-Ví dụ - (3)

Ví dụ 2.21. (Khắc phục - global)

```

n = 10
S = 0
def tong():
    global S # => S có thể thay đổi giá trị trong hàm
    i = 0
    while i <= n:
        S += i
        i += 1
    return S
print(f"Tổng các số từ 1 --> {n}: {tong()}")

```

Hàm: Hàm trong hàm

- **Hàm lồng nhau** (nested) hay **hàm con**: hàm được định nghĩa bên trong 1 hàm khác.
- ~→
 - 1 Tạo ra mã dễ đọc hơn
 - 2 Quản lý phạm vi biến một cách hiệu quả hơn.
- Điểm chính mà chúng ta cần ghi nhớ: các **hàm được bao quanh** (closure-1,2) có thể truy cập được các biến của các hàm bao quanh.



Hàm: Hàm trong hàm-Ví dụ - (1)

Ví dụ. (Hàm lồng nhau)

```
def outer_function():  
    x = 1  
    def inner_function():  
        y = 4  
        tong = x + y  
        return tong  
    return inner_function()  
  
ket_qua = outer_function()  
print(ket_qua)
```

Hàm: Hàm trong hàm-Ví dụ - (2)

Ví dụ. (Lỗi chưa tham chiếu-trong hàm con - UnboundLocalError)

```
def outer_function():  
    x = 1 # x là biến cục bộ  
    def inner_function():  
        x += 4  
        return x  
    return inner_function()  
  
ket_qua = outer_function()  
print(ket_qua)
```

Hàm: Hàm trong hàm-Ví dụ - (3)

Ví dụ. (Khắc phục - `nonlocal`)

```
def outer_function():  
    x = 1  
    def inner_function():  
        nonlocal x  
        x += 4  
        return x  
    return inner_function()  
  
ket_qua = outer_function()  
print(ket_qua)
```

Hàm: Thay đổi giá trị tham số

- Trong Python, tham số được truyền vào hàm được xem như là các tham số không thể thay đổi (immutable) $\Leftrightarrow$ nếu thay đổi giá trị của 1 tham số trong hàm, thì giá trị của tham số đó sẽ không bị ảnh hưởng bên ngoài hàm.
- Tuy nhiên, nếu tham số là 1 ĐT có khả năng thay đổi (mutable) như *danh sách*, *từ điển* $\Rightarrow$ có thể thay đổi nội dung của ĐT đó, và các thay đổi sẽ được phản ánh bên ngoài hàm.

Hàm: Thay đổi giá trị tham số-Ví dụ - (1)

Ví dụ. (Tham số không thay đổi được)

```
def modify_value(x):  
    x += 1  
    print(f"Trong hàm: {x}")  
  
number = 0  
modify_value(number)  
  
print(f"Ngoài hàm: {number}")
```

Hàm: Thay đổi giá trị tham số-Ví dụ - (2)

Ví dụ. (Tham số thay đổi được)

```
def modify_list(L):
```

```
    L[0] = 100
```

```
List = [1, 2, 3, 4, 5]
```

```
modify_list(List)
```

```
print(List)
```

Hàm: Tham số là hàm

- **Hàm callback**: hàm đóng vai trò tham số (truyền vào) trong 1 hàm khác.
- ~> Được sử dụng bên trong hàm gọi để thực hiện các tác vụ hoặc XLDL bên trong hàm gọi.



Hàm: Tham số là hàm-Ví dụ

Ví dụ. (Hàm callback)

```
def tinh_toan(a, b, ham_callback):  
    ket_qua = ham_callback(a, b)  
    return ket_qua  
  
def tinh_tong(x, y):  
    return x + y  
  
def tinh_hieu(x, y):  
    return x - y  
  
tong = tinh_toan(10, 2, tinh_tong)  
print(f"Tổng là: {tong}")  
hieu = tinh_toan(10, 2, tinh_hieu)  
print(f"Hiệu là: {hieu}")
```

Hàm: Hàm đệ quy

- **Hàm đệ quy** (recursive function): hàm mà trong quá trình thực thi nó gọi lại chính nó với 1 hoặc nhiều đối số khác nhau.
- **Cấu trúc hàm đệ quy:**
 - ① **Cơ sở** (base, *cố định* hay *neo*): hàm dừng và trả về 1 giá trị.
 - ② **Đệ quy**: hàm gọi lại chính nó với các tham số giảm đi và tiến đến phần cơ sở.



Hàm: Hàm đệ quy-Ví dụ

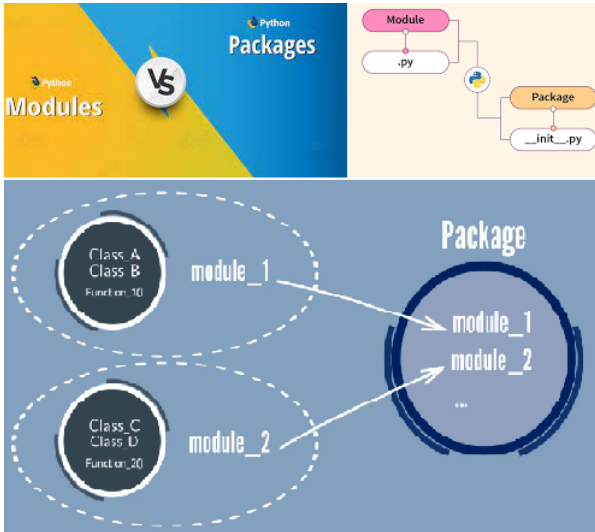
- $\leadsto$ Thường được sử dụng để giải quyết các bài toán có cấu trúc
 - ① Lặp đi lặp lại
 - ② Có thể được chia thành các phần nhỏ hơn.

Ví dụ. (Hàm đệ quy)

```
def Tinh_giai_thua(n):
    if n == 0:
        return 1 # Phần cơ sở
    else:
        x = Tinh_giai_thua(n - 1) # Phần đệ quy
        return n*x

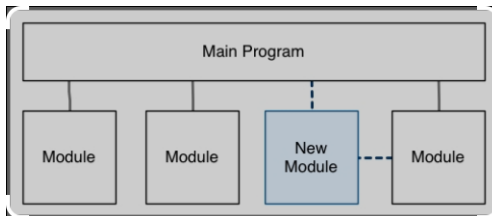
ket_qua = Tinh_giai_thua(3)
print(f"Giai thừa của 3: ", ket_qua)
```

Mô-đun và gói



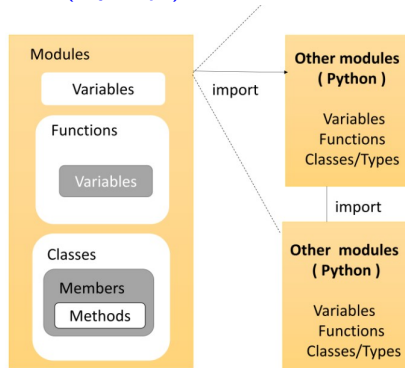
Mô-đun và gói: Mô-đun

- **Mô-đun** (module): 1 tập Python (.py) chứa các định nghĩa về *hàm, lớp, biến* và *mã Python*.
- ~→
 - ① Giúp tổ chức, tái sử dụng mã Python một cách dễ dàng.
 - ② Có thể **import** vào CT khác (để sử dụng các định nghĩa trong đó).
 - ③ Là 1 ĐT với các thuộc tính mà chúng ta có thể đặt tên tùy ý, có thể gắn kết và tham chiếu.



Mô-đun và gói: Cấu trúc mô-đun

- Một mô-đun trong Python có cấu trúc:
 - 1 *Docstrings* (tùy chọn)
 - 2 Import mô-đun khác (nếu cần)
 - 3 Khai báo *biến toàn cục* (nếu cần)
 - 4 Khai báo *hàm, lớp* (nếu cần)
 - 5 *Mã thực thi* (tùy chọn).



Mô-đun và gói: Mô-đun-Lưu ý

- ~> **Mô-đun**

- ① **Thư viện:** xây dựng sẵn các *hàm, lớp, biến ...* và có thể sử dụng cho các CT khác.
- ② **Mô-đun viết sẵn:** Python cung cấp rất nhiều.
- ③ **Mô-đun của NSD:** cho phép NLT tạo để sử dụng cho các CT của mình.



Mô-đun và gói: Mô-đun-Ví dụ

- Tạo 1 mô-đun đơn giản và sử dụng nó:

Ví dụ. (Tạo mô-đun với tên `my_module.py`)

```
my_variable = 10 # ĐN 1 biến
def say_hello(name): # ĐN 1 hàm
    return f"Xin chào, {name}!"
```

Ví dụ. (CT sử dụng `my_module.py`)

```
import my_module # Cùng thư mục
print(my_module.my_variable) # Su dụng biến
message = my_module.say_hello("Tom") # Su dụng hàm
print(message)
```

Kết quả:

10

Xin chào, Tom!

Mô-đun và gói: Mô-đun-Lưu ý

- $\leadsto$ **Mô-đun**

- 1 **Mặc định:** Python sẽ chỉ nạp

- Các mô-đun hệ thống của nó.
- Các mô-đun ở cùng cấp với tệp hiện tại.

- 2 **Nạp mô-đun ở các thư mục khác:**

- Mô-đun `sys`
- PT `.append()`

```
import sys
```

```
sys.path.append([đường dẫn chứa mô-đun cần nạp])
```

```
import <mô-đun cần nạp>
```

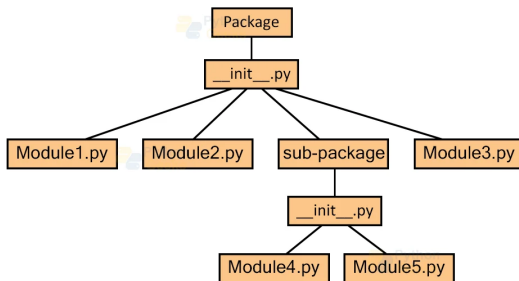
- 3 **Thay đổi dấu đường dẫn:** “\” $\Rightarrow$ “/”

Mô-đun và gói: Gói

- **Gói** (package): tổ chức các mô-đun liên quan với nhau vào trong cùng 1 thư mục.
- ~→
 - ① Quản lý mã có cấu trúc hơn.
 - ② Có thể tái sử dụng mã 1 cách có hiệu quả, đặc biệt khi dự án lớn, có nhiều mô-đun.
 - ③ Tránh xung đột giữa các mô-đun có tên giống nhau trong các dự án khác nhau.
- Tập **`__init__.py`**: phải có trong mỗi gói Python (có thể trống, hoặc chứa mã Python để khởi tạo gói).
- **Tên gói** $\equiv$ tên thư mục.

Mô-đun và gói: Cấu trúc gói

• Cấu trúc 1 gói trong Python:



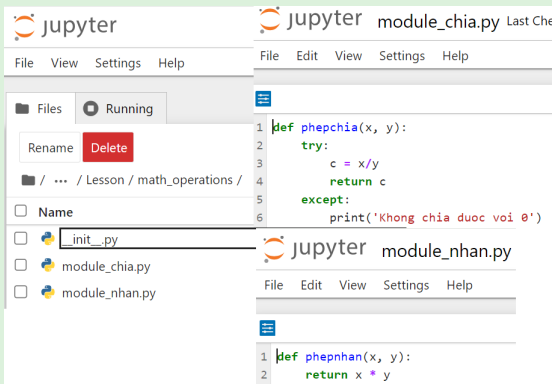
• Xây dựng 1 gói trong Python:

- ① Tạo thư mục có tên <tên gói> cần tạo (*thư mục gói*)
- ② Trong **thư mục gói** xây dựng các mô-đun cần tạo
- ③ Tạo 1 hàm `__init__.py` và lưu vào trong **thư mục gói**

Mô-đun và gói: Gói-Ví dụ - (1)

Ví dụ. (Tạo gói math_operations)

- Gói `math_operations` có 2 mô-đun `module_nhan.py`, `module_chia.py` và tệp khởi tạo `__init__.py` trông:



Mô-đun và gói: Gói-Ví dụ - (2)

Ví dụ. (tiếp theo)

- CT Main - sử dụng gói **math_operations** và các mô-đun trong nó:

```
from math_operations import module_nhan, module_chia
kq_nhan = module_nhan.phepnhan(2, 3)
kq_chia = module_nhan.phepchia(5, 1)
print(f"Kết quả nhân: {kq_nhan}")
print(f"Kết quả chia: {kq_chia}")
```

- ~ **import** gói trong 1 CT Main:
 - Các dòng lệnh trong tệp **__init__.py** sẽ được thực hiện
 - Sau đó có thể truy cập các mô-đun, biến chung từ gói đó.

Mô-đun và gói: Gói-Ví dụ - (3)

Ví dụ. (Tạo gói my_package1)

- Gói my_package1 có 2 mô-đun module1.py, module2.py và tệp khởi tạo `__init__.py` khác rỗng:

The screenshot displays the JupyterLab interface for creating a package named `my_package1`. The left sidebar shows the file explorer with the following files:

- `__init__.py`
- `module1.py`
- `module2.py`

The main area shows the code editor for `__init__.py`, `module1.py`, and `module2.py`.

__init__.py (Last Checkpoint: la:)

```
1 def common_function():
2     print("Hàm chung của gói my package")
```

module1.py

```
1 def say_hello(name):
2     return f"Hello, {name}!"
```

module2.py

```
1 my_variable = 10
```

Mô-đun và gói: Gói-Ví dụ - (4)

Ví dụ. (Tạo gói my_package2)

- Gói `my_package2` có 2 mô-đun `module1.py`, `module2.py` và tệp khởi tạo `__init__.py` khác rỗng (tổng quát):

The image displays three Jupyter Notebook windows illustrating the structure of a Python package named `my_package2`.

Window 1: File Explorer
 Shows the file structure of the package `my_package2`. The files listed are `__init__.py`, `module1.py`, and `module2.py`.

Window 2: `__init__.py`
 The content of the `__init__.py` file is shown, which imports `module1` and `module2` and defines a `common_function` that prints a message.

```

1 # my_package2/__init__.py
2
3 # Khai báo các module
4 from . import module1
5 from . import module2
6
7 # Khai báo biến:
8 a = 50 # Biến dùng chung
9
10 # Khai báo hàm:
11 def common_function():
12     print("Đây là hàm dùng chung!")
  
```

Window 3: `module1.py`
 The content of the `module1.py` file is shown, which defines a `function1` that prints a message.

```

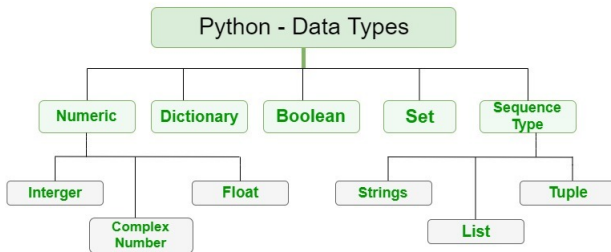
1 x1 = 20 # Biến trong mô-đun
2
3 def function1():
4     print("Đây là hàm của module1!")
  
```

Nội dung môn học

- ① Tổng quan về Python
- ② Ngữ nghĩa, cú pháp LT trong Python
- ③ Các kiểu dữ liệu phức tạp của Python
- ④ Lập trình hướng đối tượng trong Python
- ⑤ Một số thư viện chuẩn của Python
- ⑥ Tài liệu tham khảo

Chương 3. Các KDL phức tạp của Python

- 1 Kiểu dữ liệu chuỗi
- 2 Kiểu dữ liệu danh sách
- 3 Kiểu dữ liệu bộ
- 4 Kiểu dữ liệu tập hợp
- 5 Kiểu dữ liệu boolean
- 6 Kiểu dữ liệu từ điển
- 7 Kiểu dữ liệu tập



Các KDL phức tạp: KDL chuỗi-Khai báo

- **Chuỗi** (string): dãy bất kỳ các ký tự.
- **Khai báo chuỗi:**

$$\langle BC \rangle = '...' \text{ hoặc } "..."$$
- **Chuỗi rỗng** (" " hoặc " "): chuỗi không có ký tự nào.
- Các ký tự trong chuỗi là các ký tự thuộc bảng mã Unicode hoặc bảng mã ASCII $\Rightarrow$ chuỗi trong Python có hỗ trợ tiếng Việt.
- **Đánh chỉ số thứ tự các ký tự trong chuỗi:**
 - ① Theo hướng từ trái sang phải: 0, 1, ...
 - ② Theo hướng từ phải sang trái: -1, -2, ...
- $\leadsto$
 - ① Một trong những KDL cơ bản, phổ biến nhất.
 - ② Không thể thay đổi các ký tự trong biến chuỗi đã khai báo.
 - ③ Mỗi ký tự trong Python được xem như 1 chuỗi có độ dài bằng 1.

Các KDL phức tạp: KDL chuỗi-Ví dụ

Ví dụ.

```
st = "Lập trình Python"  
print(st)  
print(st[0], st[1])  
print(st[-1], st[-2])
```

Kết quả:

Lập trình Python

L

n

Các KDL phức tạp: Các thao tác trên chuỗi - (1)

- Python có rất nhiều PT, toán tử cho các thao tác (operations) trên chuỗi.
- Truy cập ký tự trong chuỗi (*index*):**

$$< BC > [index]$$

- Tính độ dài của chuỗi:**

$$len(< BC >)$$

- Nối (ghép, cộng) và lặp chuỗi:**
 - $< BC1 > + \dots + < BCn >$
 - $< BC > * n = n * < BC >$, với $n \in \mathbb{N}^*$ là số lần lặp.

Các KDL phức tạp: Các thao tác trên chuỗi - (2)

- **Phép cắt (slicing) chuỗi:**

$\langle BC \rangle [start : stop : step]$

- Trích xuất 1 chuỗi con của chuỗi $\langle BC \rangle$ từ chỉ số $start$ $\Rightarrow$ chỉ số $stop - 1$ ($start \leq stop$) với bước nhảy $step$.
- ① $\langle BC \rangle [start : stop : 1] = \langle BC \rangle [start : stop]$.
- ② $\langle BC \rangle [0 : stop] = \langle BC \rangle [: stop]$.
- ③ $\langle BC \rangle [start :]$: cắt chuỗi từ vị trí $start \Rightarrow$ cuối chuỗi.
- ④ $\langle BC \rangle [start :: step]$: cắt từ $start \Rightarrow$ hết chuỗi với bước nhảy $step$.
- ⑤ $\langle BC \rangle [::-1]$: đảo ngược chuỗi.
- Kết quả của phép cắt: trả về 1 chuỗi mới chứa các phần tử cần cắt, không làm thay đổi chuỗi ban đầu.
- $\Rightarrow$

$\langle BC \rangle [: a] + \langle BC \rangle [a :] = \langle BC \rangle [:] = \langle BC \rangle$

Các KDL phức tạp: Các thao tác trên chuỗi - (3)

- **Tách chuỗi:**

$\langle BC \rangle$.*split*(separator, maxsplit)

- Tách chuỗi thành DS các chuỗi con



- ① *separator*: ký tự phân tách, mặc định là khoảng trắng.
- ② *maxsplit* (tùy chọn): số chuỗi tối đa được tách, mặc định -1 (tách tất cả).

- **So sánh chuỗi** (comparison):

- Các toán tử so sánh: $=, ! =, <, >, <=, >=$
- Thực hiện: lần lượt so sánh các ký tự tương ứng từ trái sang phải:
 - ① Nếu gặp 1 ký tự không bằng nhau $\Rightarrow$ chuỗi nào chứa ký tự lớn hơn sẽ là chuỗi lớn hơn.
 - ② Nếu 1 chuỗi là phần đầu của chuỗi kia $\Rightarrow$ chuỗi đó là nhỏ hơn.

Các KDL phức tạp: Các thao tác trên chuỗi - (4)

- Chuyển đổi ký tự hoa, ký tự thường:

Phương thức	Thực hiện
<code>< BC >.upper()</code>	Chuyển các ký tự trong chuỗi thành <u>ký tự hoa</u>
<code>< BC >.lower()</code>	Chuyển các ký tự trong chuỗi thành <u>ký tự thường</u>
<code>< BC >.capitalize()</code>	Chuyển <u>ký tự đầu tiên</u> trong chuỗi thành <u>ký tự hoa</u> và <u>tất cả các ký tự còn lại</u> thành <u>ký tự thường</u>
<code>< BC >.title()</code>	Chuyển <u>ký tự đầu tiên của các từ</u> trong chuỗi thành <u>ký tự hoa</u>
<code>< BC >.swapcase()</code>	Chuyển các ký tự hoa thành <u>ký tự thường</u> , ký tự thường thành <u>ký tự hoa</u>

Các KDL phức tạp: Các thao tác trên chuỗi - (5)

- Kiểm tra chuỗi con:

$\langle BCC \rangle \text{ in } \langle BC \rangle$

- Trả về giá trị *True* nếu chuỗi con $\langle BCC \rangle$ có trong chuỗi $\langle BC \rangle$, ngược lại *False*.

- Đếm chuỗi con:

$\langle BC \rangle .count(\langle BCC \rangle, [start, stop])$

- Trả về số lần xuất hiện của chuỗi con $\langle BCC \rangle$ trong chuỗi $\langle BC \rangle$

↪

- 1 $[start, stop]$: phạm vi tìm kiếm trong chuỗi ($\equiv stop - 1$).
- 2 *mặc định*: toàn bộ phạm vi tìm kiếm.

Các KDL phức tạp: Các thao tác trên chuỗi - (6)

- **Tìm kiếm chuỗi:**

$\langle BC \rangle .find(\langle BCC \rangle, [start, stop])$

- Trả về chỉ số đầu tiên của chuỗi con $\langle BCC \rangle$ xuất hiện trong chuỗi $\langle BC \rangle$ hoặc -1 nếu chuỗi con không tìm thấy

~>

- 1 $[start, stop]$: phạm vi tìm kiếm trong chuỗi.
- 2 *mặc định*: toàn bộ phạm vi tìm kiếm.

~> và

$\langle BC \rangle .index(\langle BCC \rangle, [start, stop])$

- Trả về vị trí đầu tiên của chuỗi con $\langle BCC \rangle$ xuất hiện trong chuỗi $\langle BC \rangle$, hoặc ném 1 ngoại lệ **ValueError** nếu chuỗi con không tìm thấy.

Các KDL phức tạp: Các thao tác trên chuỗi - (7)

- Thay thế chuỗi con:

$\langle BC \rangle$.*replace*(*BCC_old*, *BCC_new*, [*count*])

- Trả về chuỗi mới được tạo thành từ chuỗi $\langle BC \rangle$ bằng cách thay thế tối đa *count* (tùy chọn, mặc định là tất cả) các chuỗi con $\langle BCC_old \rangle$ bằng chuỗi con $\langle BCC_new \rangle$.

- Xóa các khoảng trắng 2 đầu của chuỗi:

- $\langle BC \rangle$.*lstrip*(): xóa hết khoảng trắng (white space) ở bên trái chuỗi.
 - $\langle BC \rangle$.*rstrip*(): xóa hết khoảng trắng ở bên phải chuỗi.
 - $\langle BC \rangle$.*strip*(): xóa hết khoảng trắng ở cả 2 đầu chuỗi.
- ↪ Thường được dùng khi XLDL người dùng nhập vào.

Các KDL phức tạp: Các thao tác trên chuỗi - (8)

- **Căn chỉnh (align) chuỗi:**
 - ① `< BC > .ljust(width)`: Căn chỉnh bên trái, bằng cách thêm số khoảng trắng *width* vào bên phải chuỗi.
 - ② `< BC > .rjust(width)`: Căn chỉnh bên phải, bằng cách thêm số khoảng trắng *width* vào bên trái chuỗi.
 - ③ `< BC > .center(width)`: Căn chỉnh chuỗi vào giữa, bằng cách thêm số khoảng trắng *width* vào đều 2 bên chuỗi.
- ~ Thường được dùng để định dạng chuỗi theo cách chúng ta muốn.

Các KDL phức tạp: Các thao tác trên chuỗi - (9)

- **Kiểm tra tính chất (*True, False*) của chuỗi (*is_*):**
 - ➊ $\langle BC \rangle$ **.isspace()**: chỉ chứa các ký tự trắng?
 - ➋ $\langle BC \rangle$ **.isdigit()**: chỉ chứa các chữ (ký tự) số?
 - ➌ $\langle BC \rangle$ **.islower()**: chỉ chứa các chữ thường?
 - ➍ $\langle BC \rangle$ **.isupper()**: chỉ chứa các chữ hoa?
 - ➎ $\langle BC \rangle$ **.isalpha()**: chỉ chứa các chữ ('a', 'b', ..., 'z' và 'A', 'B', ..., 'Z')?
 - ➏ $\langle BC \rangle$ **.isalnum()**: chỉ chứa các chữ hoặc số?
 - ➐ $\langle BC \rangle$ **.istitle()** : chỉ chứa các từ có chữ đầu là chữ hoa?
- $\leadsto$ Thường được dùng để kiểm tra tính chất của 1 chuỗi trước khi xử lý nó.

Các KDL phức tạp: Các thao tác trên chuỗi - (10)

- **Hàm** max, min:
 - ① **max**(< BC >): trả về ký tự có giá trị lớn nhất trong chuỗi dựa trên giá trị ASCII của các ký tự
 - ② **min**(< BC >): trả về ký tự có giá trị nhỏ nhất trong chuỗi dựa trên giá trị ASCII của các ký tự.
 - **Hàm** eval:
 - **eval**(< BC >): tính 1 biểu thức Python được biểu diễn dưới dạng chuỗi.
- ~>
- ① Hàm rất mạnh $\Rightarrow$ cho phép thực thi bất kỳ mã Python nào được cung cấp trong chuỗi, kể cả những mã có thể gây lỗi hoặc nguy hiểm $\Rightarrow$ gây ra lỗ hổng bảo mật hoặc lỗi cho ứng dụng nếu không được sử dụng đúng cách.
 - ② Hạn chế sử dụng hàm này khi không cần thiết.

Các KDL phức tạp: KDL danh sách-Khai báo

- **Danh sách** (list): tập hợp các phần tử (bất kỳ).
- **Khai báo DS:**

$$< BDS > = [x_1, x_2, \dots, x_n]$$

- Các phần tử trong DS của Python có thể giống nhau, khác kiểu và thậm chí là 1 DS.
- **DS rỗng** (`[]`): DS không có phần tử nào.
- **Đánh chỉ số thứ tự các phần tử trong DS:** tương tự chuỗi.
- $\leadsto$
 - Một trong những KDL cực mạnh của Python.
 - Một dạng mở rộng của KDL mảng (array) trong các NNLT khác (*C, Java ...*).

Các KDL phức tạp: KDL danh sách-Ví dụ

Ví dụ. (DS)

```
L1 = [ ]
```

```
L2 = [2, 1, 3.5, 2, 5, 'Toán', ['a', 'b'], 'Văn']
```

```
print("List 1 = ", L1)
```

```
print("List 2 = ", L2)
```

```
print("Phần tử thứ 1 từ trái sang của List 2: ", L2[0])
```

```
print("Phần tử thứ 1 từ phải sang của List 2: ", L2[-1])
```

Kết quả:

```
List 1 = [ ]
```

```
List 2 = [2, 1, 3.5, 2, 5, 'Toán', ['a', 'b'], 'Văn']
```

```
Phần tử thứ 1 từ trái sang của List 2: 2
```

```
Phần tử thứ 1 từ phải sang của List 2: Văn
```

Các KDL phức tạp: Các thao tác trên DS- 1

- Python hỗ trợ rất nhiều thao tác trên DS.
- **Truy cập các phần tử trong DS:**

$\langle BDS \rangle [index]$

- **Lấy số phần tử trong DS:**

$len(\langle BDS \rangle)$

- **Cập nhật giá trị phần tử trong DS:**

$\langle BDS \rangle [index] = \langle \text{giá trị mới} \rangle$

↪ Đây là điểm khác so với chuỗi.

Các KDL phức tạp: Các thao tác trên DS- 2

- Cắt (slice) DS:**

$$\langle BDS \rangle = [start : stop : step]$$

- Trích xuất 1 phần của DS theo khoảng chỉ định: từ vị trí *start* đến vị trí *stop* – 1 với bước nhảy *step*
- ① $\langle BDS \rangle [start : stop : 1] = \langle BDS \rangle [start : stop]$
- ② $\langle BDS \rangle [0 : stop] = \langle BDS \rangle [: stop]$
- ③ $\langle BDS \rangle [start :]$: cắt DS từ vị trí *start* đến hết DS
- ④ $\langle BDS \rangle [start :: step]$: cắt từ *start* đến hết DS với *step*
- ⑤ $\langle BDS \rangle [::-1]$: đảo ngược DS
- Kết quả của phép cắt: trả về 1 DS mới chứa các phần tử cần cắt, không làm thay đổi DS ban đầu.
- Nối (ghép) và lặp DS:**
 - ① $\langle BDS1 \rangle + \dots + \langle BDSn \rangle$
 - ② $\langle BDS \rangle * n = n * \langle BDS \rangle$, với $n \in \mathbb{N}^*$ là số lần lặp.

Các KDL phức tạp: Các thao tác trên DS- 3

- Thêm vào cuối DS:

$\langle BDS \rangle .append(x)$

- x : phần tử, DS.
- Mở rộng DS:

$\langle BDS \rangle .extend(\langle BDS' \rangle)$

- $\leadsto append() \neq extend()$.

Các KDL phức tạp: Các thao tác trên DS- 4

- Kiểm tra sự tồn tại phần tử trong DS (*True, False*) :

$\langle x \rangle \text{ *in* } \langle BDS \rangle \quad \langle x \rangle \text{ *not in* } \langle BDS \rangle$

- Tìm 1 phần tử trong DS:

$\langle BDS \rangle .\text{*index*}(x)$

- Trả về chỉ số của pt đầu tiên bằng x trong DS.
- Không tồn tại phần tử bằng x trong DS $\Rightarrow$ ngoại lệ **ValueError**. Trường hợp này, có thể kết hợp sử dụng toán tử *in* và PT *index()*.

- Đếm 1 phần tử trong DS:

$\langle BDS \rangle .\text{*count*}(x)$

Các KDL phức tạp: Các thao tác trên DS- 5

- Đảo ngược DS:

$\langle BDS \rangle .reverse()$

- Tính tổng các phần tử trong DS:

$sum(\langle BDS \rangle)$

- Hàm max , min trên DS:

- ① $max(\langle BDS \rangle)$: Trả về phần tử có GTLN trong DS
 - ② $min(\langle BDS \rangle)$: Trả về phần tử có GTNN trong DS
- 2 hàm rất hữu ích khi cần tìm GTLN, GTNN trong DS để thực hiện các tính toán hoặc kiểm tra.

Các KDL phức tạp: Một số vấn đề nâng cao DS-1

- **Tạo 1 DS:**

① $\langle BDS \rangle .append(\langle x \rangle)$

- Tạo DS bằng cách thêm từng phần tử vào DS.

② $[\langle value \rangle \text{ for } \langle x \rangle \text{ in } \langle \text{đối tượng} \rangle]$

- Tạo DS gồm các phần tử là giá trị của *value* (giá trị hoặc biểu thức) khi phần tử lần lượt nhận giá trị của (tập) đối tượng.

③ $[\langle value \rangle \text{ for } \langle x \rangle \text{ in } \langle \text{đối tượng} \rangle \text{ if } \langle \text{điều kiện} \rangle]$

- Tương tự như trên nhưng các phần tử phải thỏa thêm điều kiện.

④ Hai vòng lặp **for** lồng nhau: tạo DS với các phần tử cũng là DS (DS 2 chiều).

Các KDL phức tạp: Một số vấn đề nâng cao DS-2

Ví dụ. (Tạo DS bằng *for*)

```
L1 = [i for i in range(5)]  
L2 = [i*i for i in range(5)]  
L3 = [0 for i in range(5)]  
L4 = [5*i for i in (1,2,3) if i%2 == 0]  
print(f"List 1 = {L1}")  
print(f"List 2 = {L2}")  
print(f"List 3 = {L3}")  
print(f"List 4 = {L4}")
```

Kết quả:

```
List 1 = [0, 1, 2, 3, 4]  
List 2 = [0, 1, 4, 9, 16]  
List 3 = [0, 0, 0, 0, 0]  
List 4 = [10]
```

Các KDL phức tạp: Một số vấn đề nâng cao DS-3

Ví dụ. (Tạo DS bằng 2 *for*)

```
L = [[i, j] for i in (1, 2) for j in range(3)]  
print("List = ", L)
```

Kết quả:

```
List = [[1, 0], [1, 1], [1, 2], [2, 0], [2, 1], [2, 2]]
```

Ví dụ. (Tạo DS bằng *for* và *append*)

```
L = []  
for i in range(10):  
    L.append(i)  
print("List = ", L)
```

Kết quả:

```
List = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Các KDL phức tạp: Một số vấn đề nâng cao DS-4

- Chèn phần tử vào DS:

$\langle BDS \rangle .insert(i, x)$

- Chèn x vào vị trí có chỉ số i trong DS. Trường hợp i lớn hơn số phần tử của DS, thì chèn x vào cuối DS.
- Xóa phần tử khỏi DS:

① PT:

- ① $\langle BDS \rangle .remove(x)$: Xóa phần tử x gặp đầu tiên trong DS. Trường hợp nếu DS không có x , thì Python ném ra 1 ngoại lệ **ValueError**.
- ② $\langle BDS \rangle .pop()$: Xóa phần tử cuối cùng của DS.
- ③ $\langle BDS \rangle .clear()$: Xóa tất cả các phần tử trong DS.

② Hàm `del()`:

- ① $del \langle BDS \rangle [index]$: Xóa phần tử có chỉ số $[index]$.
- ② $del \langle BDS \rangle [start : stop : step]$: Xóa các phần tử trong phạm vi $[start : stop - 1]$ với bước nhảy là $step$ (mặc định $step = 1$).

Các KDL phức tạp: Một số vấn đề nâng cao DS-5

- **Sắp xếp DS:**

- ① **PT `.sort()`:**

- ① `< BDS > .sort()`: Thứ tự tăng.

- ② `< BDS > .sort(reverse = True)`: Thứ tự giảm.

- DS gốc sẽ bị thay đổi sau khi sắp xếp bằng PT này.

- ② **Hàm `sorted()`:**

- ① `sorted(< BDS >)`: Sắp DS theo thứ tự tăng dần.

- ② `sorted(< BDS >, reverse = True)`: Sắp DS theo thứ tự giảm dần.

- Hàm này sẽ tạo ra 1 DS mới sau khi sắp xếp và không làm thay đổi DS gốc.

Các KDL phức tạp: Một số vấn đề nâng cao DS-6

- **Ma trận - DS 2 chiều:**
 - **Ma trận (Python):** DS mà mỗi phần tử của nó là 1 DS
- biểu diễn 1 dòng của ma trận.

Ví dụ.

- $[[1, 0, 1], [0, 0, 1]]$ - DS chứa các DS con biểu diễn cho ma trận:

1	0	1
0	0	1

- Đánh chỉ số dòng, cột trong ma trận: bắt đầu từ 0.
- Truy cập 1 phần tử trong ma trận:

$\langle BDS \rangle [\text{chỉ số dòng}][\text{chỉ số cột}]$

Các KDL phức tạp: Một số vấn đề nâng cao DS-7

Ví dụ. (Duyệt ma trận)

matrix = [[1, 0, 1], [0, 0, 1]] # GS đã có ma trận

print(matrix) # Cách 1

for x **in** matrix: # Cách 2: 1 for

print(x)

for i **in** range(2): # Cách 3: theo chỉ số

for j **in** range(3):

print(matrix[i][j], **end** = " ")

print()

Kết quả:

[[1, 0, 1], [0, 0, 1]]

[1, 0, 1] [0, 0, 1]

1 0 1

0 0 1

Các KDL phức tạp: Một số vấn đề nâng cao DS-8

Ví dụ. (tiếp theo)

```
matrix = [[1, 0, 1], [0, 0, 1]]
```

Cách 4: duyệt từng phần tử theo hàng, cột

```
for row in matrix:
```

```
    for element in row:
```

```
        print(element, end = ' ')
```

```
    print()
```

Kết quả:

```
1 0 1
```

```
0 0 1
```

- DS 2 chiều là 1 cách phổ biến để biểu diễn ma trận trong Python và thực hiện các phép toán ma trận cơ bản.

Các KDL phức tạp: Một số vấn đề nâng cao DS-9

Ví dụ. (Tạo ma trận)

```
m = int(input("Nhập số hàng ma trận: "))  
n = int(input("Nhập số cột ma trận: "))
```

Cách 1:

```
matrix = []
```

```
for i in range(m):
```

```
    row = []
```

```
    for j in range(n):
```

```
        a = int(input(f"a{i + 1, j + 1} = "))
```

```
        row.append(a)
```

```
    matrix.append(row)
```

Cách 2:

```
matrix = [[int(input(f"a{i + 1, j + 1}")) for j in range(n)]
```

```
for i in range(m)]
```

Các KDL phức tạp: Một số vấn đề nâng cao DS-10

- **DS của các DS:**
 - Ma trận là trường hợp riêng của DS mà các phần tử của nó là các DS
 - Mỗi phần tử trong DS là các DS mà số phần tử của nó có thể khác nhau.

Ví dụ. (DS của DS)

```
L1 = [1, 2, 3]
L2 = ['Programming', 'Python']
L3 = [2023]
L = [L1, L2]
L.append(L3)
print(f"DS là {L}")
```

Kết quả:

DS là: `[[1, 2, 3], ['Programming', 'Python'], [2023]]`

Các KDL phức tạp: Một số vấn đề nâng cao DS-11

- So sánh 2 DS:

- ① Lần lượt so sánh các phần tử từ chỉ số 0 đến hết DS
- ② Khi gặp 2 phần tử tương ứng khác nhau, thì DS nào chứa phần tử lớn hơn, DS đó lớn hơn
- ③ Các phép toán so sánh trên DS là: $==$, $<$, $<=$, $>$, $>=$

Ví dụ. (Các phép toán SS)

$L1 = [1, 2, 3]$; $L2 = [1, 3, 2]$; $L3 = [1, 2, 3, 4]$

```
print(f"1. {L1 == L2}")
```

```
print(f"2. {L1 < L2}")
```

```
print(f"3. {L1 <= L2}")
```

```
print(f"4. {L1 > L3}")
```

```
print(f"5. {L1 >= L3}")
```

Kết quả:

1. False, 2. True, 3. True, 4. False, 5. False

Các KDL phức tạp: KDL bộ-Khai báo

- **Bộ** (tuple): 1 tập các phần tử được sắp thứ tự.
- **Khai báo 1 bộ trong Python:**

$$\langle BB \rangle = (x_1, x_2, \dots, x_n)$$

- Trường hợp nếu bộ chỉ có 1 phần tử:

$$\langle BB \rangle = (\langle x \rangle,)$$

- **Bộ rỗng** ($()$): bộ không có phần tử nào.
- **Đánh chỉ số thứ tự các ký tự trong bộ:** tương tự chuỗi, DS.
- $\leadsto$
 - Bộ có thể chứa nhiều KDL: số nguyên, số thực, chuỗi và thậm chí là DS hoặc các bộ khác.

Các KDL phức tạp: KDL bộ-Ví dụ



- Khác với DS, bộ là KDL không thay đổi (immutable): 1 bộ sau khi được tạo thì không thể *thêm*, *xóa* hoặc *thay đổi giá trị* các phần tử trong bộ
- $\Rightarrow$ Bộ rất thích hợp khi chúng ta muốn bảo đảm tính không thay đổi của DL.

Ví dụ. (Bộ)

t1 = (1, 2, "Python", 3.5)

t2 = (5,)

t3 = (5) # không phải bộ

Kết quả:

(1, 2, 'Python', 3.5)

(5,)

5

Các KDL phức tạp: Các thao tác trên bộ-1

- Truy cập các phần tử trong bộ:

$$\langle BB \rangle [index]$$

- Lấy số phần tử trong bộ:

$$len(\langle BB \rangle)$$

- Nối (ghép) và lặp bộ:

- 1 $\langle BB1 \rangle + \dots + \langle BBn \rangle.$

- 2 $\langle BB \rangle * n = n * \langle BB \rangle,$ với $n \in \mathbb{N}^*$ là số lần lặp.

- KDL bộ không cho phép thêm phần tử. Tuy nhiên, có thể dùng nối bộ để thêm 1 phần tử mới (tạo ra bộ mới).

Các KDL phức tạp: Các thao tác trên bộ-2

- **Phép cắt (slicing) bộ:**

$$< BB > = [start : stop : step]$$

- Trích xuất 1 phần của bộ theo khoảng chỉ định.
- ① $< BB > [start : stop : 1] = < BB > [start : stop]$
- ② $< BB > [0 : stop] = < BB > [: stop]$
- ③ $< BB > [start :]$: cắt bộ từ *start* đến hết của bộ
- ④ $< BB > [start :: step]$: cắt từ *start* đến hết bộ với *step*
- ⑤ $< BB > [::-1]$: đảo ngược bộ
- Kết quả của phép cắt: trả về 1 bộ mới chứa các phần tử cần cắt, không làm thay đổi bộ ban đầu.

- **So sánh các bộ:**

- Tương tự như DS
- Các phép toán so sánh trên bộ: $==, <, <=, >, >=$

Các KDL phức tạp: Các thao tác trên bộ-3

- **Hàm `tuple()`:**
 - **`tuple(iterable)`:** Tạo 1 bộ từ DL có thể lặp lại (*iterable*). Trường hợp không truyền đối số thì hàm tạo 1 bộ trống
 - Hàm này rất hữu ích khi muốn chuyển đổi DL từ định dạng *iterable* sang bộ để đảm bảo tính không thay đổi của DL
 - *iterable*: chuỗi, DS, bộ (khác), tập hợp, từ điển, *range()*.
- **Packing và Unpacking:**
 - ① **Packing:** đóng gói nhiều giá trị thành 1 bộ mà không cần đặt biến cho từng giá trị
 - ② **Unpacking:** trích xuất các giá trị từ 1 bộ và gán chúng cho các biến riêng lẻ
 - **Unpacking** cũng có thể được thực hiện cho: DS hoặc bất kỳ *iterable* nào khác
 - **Packing và Unpacking** giúp làm việc với bộ 1 cách linh hoạt và tiện lợi, đặc biệt khi muốn trao đổi giữa các giá trị 1 cách nhanh chóng.

Các KDL phức tạp: Các thao tác trên bộ

- **Hàm max, min trên bộ:**
 - ① **max**(< BB >): Trả về phần tử có GTLN trong bộ
 - ② **min**(< BB >): Trả về phần tử có GTNN trong bộ.
- **Kiểm tra sự tồn tại phần tử thuộc bộ (True, False):**

< x > **in** < BB > < x > **not in** < BB >

- **Tìm 1 phần tử trong bộ:**

< BB > **.index**(< x >)

- Trả về chỉ số của phần tử đầu tiên bằng x trong bộ
- Khi không tồn tại phần tử bằng x trong bộ $\Rightarrow$ ném ra 1 ngoại lệ **ValueError**.
- **Đếm 1 phần tử trong bộ:**

< BB > **.count**(< x >)

Các KDL phức tạp: KDL tập hợp-Khai báo

- **Tập hợp** (set): 1 tập các phần tử (giá trị) duy nhất, không sắp thứ tự và do đó không index.
- **Khai báo 1 tập hợp trong Python:**

$$< BTH > = \{x_1, x_2, \dots, x_n\}.$$

- **TH rỗng** ($\{\}$): TH không có phần tử nào.
- $\leadsto$
 - KDL có thể thay đổi được nội dung (mutable).
 - KDL linh hoạt và mạnh mẽ, thường được sử dụng để thực hiện các phép toán tập hợp giống như trong toán học. Đây là 1 điểm mạnh của Python.

Các KDL phức tạp: KDL tập hợp-Ví dụ

Ví dụ. (TH)

```
s = {1, 0, 2, 5, 'Python', 2, 3}
empty_set = set() # tạo tập rỗng
print(s) # Python sẽ loại bỏ các giá trị trùng lặp
print(empty_set)
```

Kết quả:

```
{0, 1, 2, 3, 5, 'Python'}
set()
```

Các KDL phức tạp: Các thao tác trên tập hợp-1

- **Tạo 1 TH:**

- ① **Liệt kê trực tiếp:** như định nghĩa
- ② Hàm **set()**: kết hợp với DS, chuỗi và PT **add()**
- ③ Vòng lặp **for**: tương tự như tạo DS.

Ví dụ. (Tạo TH)

```
s1 = set([1, 1, 2, 3, 4, 5])
s2 = set("abca")
s3 = set() ; s3.add(1) ; s3.add(5)
s4 = {x for x in range(10)}
```

Kết quả:

```
{1, 2, 3, 4, 5}
```

```
{'c', 'b', 'a'}
```

```
{1, 5}
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Các KDL phức tạp: Các thao tác trên tập hợp-2

- Lấy số phần tử của TH:

$len(< BTH >)$

- Kiểm tra sự tồn tại phần tử trong TH (*True, False*):

$< x > \text{ in } < BTH >$

$< x > \text{ not in } < BTH >$

- Thêm phần tử vào TH:

① $< BTH > .add(< x >)$: Thêm 1 phần tử

② $< BTH > .update(< iterable >)$: Thêm nhiều phần tử từ *iterable* vào TH.

Các KDL phức tạp: Các thao tác trên tập hợp-3

- **Xóa phần tử ra khỏi TH:**

- ① $\langle BTH \rangle .discard(\langle x \rangle)$: Xóa 1 phần tử
- ② $\langle BTH \rangle .remove(\langle x \rangle)$: Xóa 1 phần tử ra khỏi TH, nếu phần tử không tồn tại thì ném ra 1 ngoại lệ **KeyError**
- ③ $\langle BTH \rangle .pop()$: Xóa và trả về 1 phần tử NN từ TH
- ④ $\langle BTH \rangle .clear()$: Xóa tất cả phần tử.

- **Phép hợp:**

- ① $\langle BTH1 \rangle .union(\langle BTH2 \rangle)$: Hợp 2 TH
- ② $\langle BTH1 \rangle \mid \langle BTH2 \rangle$.

- **Phép giao:**

- ① $\langle BTH1 \rangle .intersection(\langle BTH2 \rangle)$: Giao 2 TH
- ② $\langle BTH1 \rangle \& \langle BTH2 \rangle$.

Các KDL phức tạp: Các thao tác trên tập hợp-4

- **Phép hiệu:**

- ① $\langle BTH1 \rangle .difference(\langle BTH2 \rangle)$: Hiệu của 2 TH
- ② $\langle BTH1 \rangle - \langle BTH2 \rangle$.

- **Phép hiệu đối xứng:**

- ① $\langle BTH1 \rangle .symmetric_difference(\langle BTH2 \rangle)$: Hiệu đối xứng của 2 TH
- ② $\langle BTH1 \rangle ^\wedge \langle BTH2 \rangle$.

- **Các phép toán đối với tập con (*True*, *False*):**

- ① So sánh bằng ($==$): $\langle BTH1 \rangle == \langle BTH2 \rangle$
- ② Kiểm tra tập con:
 - ① $\langle BTH1 \rangle .issubset(\langle BTH2 \rangle)$
 - ② $\langle BTH1 \rangle <= \langle BTH2 \rangle$.
- ③ Kiểm tra tập con thật sự:
 - ① $\langle BTH1 \rangle .issubset(\langle BTH2 \rangle) \text{ and } \langle BTH1 \rangle != \langle BTH2 \rangle$
 - ② $\langle BTH1 \rangle < \langle BTH2 \rangle$.

Các KDL phức tạp: Các thao tác trên tập hợp-5

- Kiểm tra các điều kiện trong 1 TH:
 - ① ***all()***: Hàm trả về *True* nếu tất cả các phần tử trong TH thoả điều kiện, ngược lại trả về *False*.
 - ② ***any()***: Hàm trả về *True* nếu ít nhất 1 phần tử trong TH thoả mãn điều kiện, ngược lại trả về *False*.

Các KDL phức tạp: KDL Boolean-1

- **Boolean** (*bool*, *logic*): biểu diễn 2 giá trị *True* (đúng) và *False* (sai).
 - Thường được sử dụng trong: *các biểu thức logic*, *cấu trúc điều kiện* và *các loại kiểm tra trong LT*.
- **Chuyển đổi sang KDL boolean trong Python:**

bool(< value >)

- Chuyển đổi 1 giá trị hoặc biểu thức thành KDL boolean:
 - ① *True*: nếu giá trị là số khác 0 hay *iterable* khác rỗng hay biểu thức là đúng
 - ② *False*: nếu giá trị là số bằng 0 hay *iterable* rỗng hay *None* hay biểu thức là sai.

Các KDL phức tạp: KDL Boolean-2

- Các phép toán trên kiểu DL boolean trong Python:**

- Các toán tử so sánh:**

- $==, !=, >, <, >=, <=$

- 2 toán hạng trong các toán tử này phải có cùng kiểu.

- Các toán tử boolean:**

- and, or, not*

- Bảng chân trị:

AND			OR			NOT	
A	B	A AND B	A	B	A OR B	A	NOT A
True	True	True	True	True	True	True	False
True	False	False	True	False	True	False	True
False	True	False	False	True	True		
False	False	False	False	False	False		

- Toán tử *in***

- Toán tử *is***

Các KDL phức tạp: KDL Từ điển-Khai báo

- **Từ điển** (dictionary, kiểu ánh xạ-mapping): tập không có thứ tự của các cặp **khóa-giá trị** (key-value), trong đó mỗi khóa phải là duy nhất.
- **Khai báo 1 TĐ trong Python:**

$$< BTĐ > = \{ < key_1 : value_1 >, \dots, < key_n : value_n > \}$$

- **Lưu trữ và truy xuất các giá trị trong TĐ:** dựa trên khóa (thay vì chỉ số như DS).
- $\leadsto$
 - KDL phổ biến và mạnh trong Python, còn được gọi là **KDL ánh xạ**
 - Thường được sử dụng rộng rãi trong lưu trữ và quản lý thông tin phức tạp một cách có hiệu quả.

Các KDL phức tạp: KDL Từ điển-Ví dụ

- Các khóa trong TĐ là không thay đổi $\Rightarrow$
 - Khoá: chỉ có thể sử dụng các KDL không thay đổi như chuỗi hoặc số
 - Giá trị: dùng KDL tùy ý.

Ví dụ. (TĐ)

$D = \{1:10.5, 2:\text{"Python"}, \text{"age"}:9\}$: TĐ gồm 3 phần tử với khóa và giá trị:

Khóa	Giá trị
1	10.5
2	"Python"
"age"	9

Các KDL phức tạp: Các thao tác trên TĐ-1

- Truy cập các phần tử trong TĐ:

$\langle BTĐ \rangle [key]$

- Lấy số phần tử trong TĐ:

$len(\langle BTĐ \rangle)$

- Thêm phần tử mới vào TĐ:

$\langle BTĐ \rangle [new\ key] = \langle \text{giá trị} \rangle$

Các KDL phức tạp: Các thao tác trên TĐ-2

- Cập nhật giá trị của khóa trong TĐ:

① $\langle BTĐ \rangle [key] = \langle \text{giá trị mới} \rangle$

② $\langle BTĐ \rangle .update(\langle BTĐ_other \rangle)$: cập nhật giá trị của nhiều khóa trong $\langle BTĐ \rangle$ từ $\langle BTĐ_other \rangle$

- Lấy DS các đối tượng trong TĐ:

① Khóa:

$\langle BTĐ \rangle .keys()$

② Giá trị:

$\langle BTĐ \rangle .values()$

③ Khóa-giá trị:

$\langle BTĐ \rangle .items()$

- Sao chép TĐ:

$\langle BTĐ \rangle .copy()$

Các KDL phức tạp: Các thao tác trên TĐ-3

- Duyệt TĐ: *for* + *print*

① Khóa:

for < BK > *in* < BTD >

② Giá trị:

for < BGT > *in* < BTD > *.values()*

③ Khóa-giá trị:

for < BK, BGT > *in* < BTD > *.items()*

- Kiểm tra sự tồn tại khóa trong TĐ (*True*, *False*):

< key > *in* < BTD > < x > *not in* < BTH >

Các KDL phức tạp: Các thao tác trên TĐ-4

- **Xóa phần tử thuộc TĐ:**

- ① `< BTD > .pop(< key >)`: Xóa 1 phần tử
- ② `< BTD > .popitem()`: Xóa 1 phần tử được đưa vào TĐ lần sau cùng
- ③ `< BTD > .clear()`: Xóa tất cả phần tử (TĐ rỗng)
- ④ `del(< BTD > [< key >])` $\equiv$ (1)
- ⑤ `del(< BTD >)` : TĐ sẽ không tồn tại.

- **Lấy giá trị của 1 khóa trong TĐ:**

`< BTD > .get(< key > [, default])`

- PT này trả về giá trị *None* nếu TĐ không có khóa. Trường hợp muốn thay *None*, dùng thêm tùy chọn *default*
- PT giúp tránh lỗi khi cố gắng truy cập 1 khóa không tồn tại trong TĐ.

Các KDL phức tạp: KDL tệp

- **Tệp** (file): KDL dùng để làm việc với các tệp tin (trên MT) như văn bản, hình ảnh, âm thanh, hoặc bất kỳ loại DL nào khác được lưu trữ trên ổ đĩa.
- Python cung cấp nhiều chức năng và PT để làm việc với tệp: đọc, ghi, tạo, xóa và di chuyển tệp.
- Có 3 chế độ (mở tệp) làm việc với tệp:
 - **r**: đọc
 - **w**: ghi
 - **a**: nối thêm (append).
- **Biến tệp**: để an toàn về DL trong tệp $\Rightarrow$ hệ điều hành yêu cầu các CT làm việc với tệp thông qua biến tệp.

Các KDL phức tạp: Các thao tác trên tệp-1



Các KDL phức tạp: Các thao tác trên tệp-2

- **Mở tệp để đọc:**

$\langle BT \rangle = \text{open}(\langle \text{"tên tệp"} \rangle, \text{"r"})$

with open($\langle \text{"tên tệp"} \rangle, \text{"r"} \rangle$) *as* $\langle BT \rangle$:

- Nếu tệp không tồn tại $\Rightarrow$ CT sẽ ném ra ngoại lệ **FileNotFoundException**.
- **Đọc tệp:**
 - ① $\langle BT \rangle .\text{read}()$: Đọc tất cả ký tự trong tệp (kể cả các ký tự xuống dòng) và lưu chúng thành 1 chuỗi ký tự
 - ② $\langle BT \rangle .\text{read}(n)$: Đọc n byte tính từ vị trí con trỏ tệp
 - ③ $\langle BT \rangle .\text{readline}()$: Đọc từng dòng của tệp
 - ④ $\langle BT \rangle .\text{readlines}()$: Đọc cả tệp và lưu chúng thành 1 DS, mỗi phần tử của DS là 1 dòng (gồm cả ký tự xuống dòng)
- $\leadsto$ Khi các ký tự được mã hóa theo bảng mã ASCII, thì mỗi ký tự được tính là 1 byte.

Các KDL phức tạp: Các thao tác trên tệp-3

- **Mở tệp để ghi:**

$\langle BT \rangle = \text{open}(\langle \text{"tên tệp"} \rangle, \text{"w"})$

with open($\langle \text{"tên tệp"} \rangle, \text{"w"} \rangle$) *as* $\langle BT \rangle$:

- Tạo ra 1 tệp để ghi DL vào
- Ban đầu tệp chưa có DL. Trường hợp nếu tệp đã tồn tại $\Rightarrow$ DL cũ sẽ bị xóa hết
- Tên tệp có thể gồm cả đường dẫn.
- **Ghi vào tệp:**
 - ① $\langle BT \rangle .\text{write}(\langle \text{Chuỗi} \rangle)$: Ghi và không xuống dòng
 - ② $\langle BT \rangle .\text{writelines}(\langle \text{Đối tượng} \rangle)$: Đối tượng (*chuỗi*, *DS*, *tập hợp*) phải là các dãy ký tự
 - ③ **print**(Nội dung ghi, *file* = $\langle BT \rangle$): Nội dung ghi có thể chữ hoặc số.

Các KDL phức tạp: Các thao tác trên tệp-4

- **Mở tệp để ghi thêm:**

$\langle BT \rangle = \text{open}(\langle \text{"tên tệp"} \rangle, \text{"a"})$

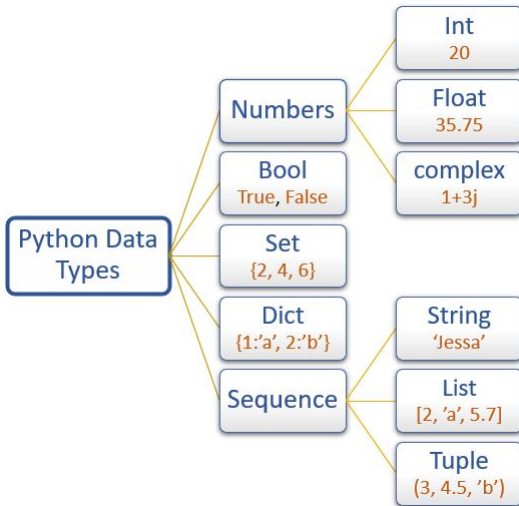
- Nếu tệp không tồn tại $\Rightarrow$ CT sẽ tự động tạo ra 1 tệp chưa có nội dung để ghi nội dung mới vào.

- **Đóng tệp:**

$\langle BT \rangle .\text{close}()$

- $\leadsto$ Đóng tệp sau khi thao tác với tệp xong.

Các kiểu DL phức tạp của Python

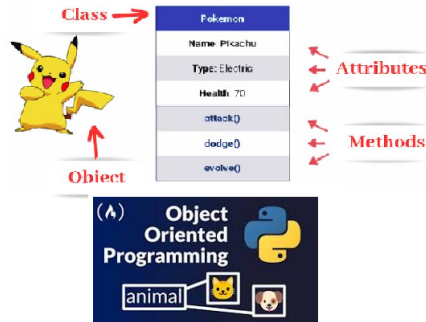


Nội dung môn học

- ① Tổng quan về Python
- ② Ngữ nghĩa, cú pháp LT trong Python
- ③ Các kiểu dữ liệu phức tạp của Python
- ④ Lập trình hướng đối tượng trong Python**
- ⑤ Một số thư viện chuẩn của Python
- ⑥ Tài liệu tham khảo

Chương 4. Lập trình hướng đối tượng trong Python

- 1 Lớp
- 2 Sự kế thừa
- 3 Các thao tác trên lớp



Lập trình HĐT: Lớp-Khai báo

- **Lớp** (Class): 1 thực thể bao gồm 2 thành phần chính
 - ① **Thuộc tính**: là các biến DL, dùng để lưu trữ DL và thông tin để mô tả thực thể đó.
 - ② **Phương thức**: là các hàm, được dùng để thực hiện các hành động hoặc chức năng cụ thể đối với các ĐT (objects) thuộc lớp đó dựa trên DL được lưu trữ trong các thuộc tính.
- $\leadsto$
 - KDL tùy chỉnh
 - Được sử dụng trong LTHĐT để tạo ra nhiều ĐT dựa trên 1 mô hình hoặc khuôn mẫu cụ thể.
- **Khai báo 1 lớp trong Python:**

class <tên lớp>:
 <Định nghĩa *thuộc tính* và *PT*>
- **Tạo 1 ĐT kiểu lớp:**

<ĐT>=<tên lớp> ([các tham số])

Lập trình HDT: Lớp-Ví dụ

- Xây dựng 1 lớp **Rectangle**:
 - Lưu các thông tin về 1 hình CN (thuộc tính)
 - Cách tính diện tích của hình CN đó (PT).

Ví dụ 4.1. (Lớp)

```
class Rectangle(): # ĐN 1 lớp
    length = 8 # ĐN các thuộc tính
    width = 5
    def area(self): # ĐN 1 PT
        S = self.length*self.width
        return S
p = Rectangle() # Tạo 1 ĐT
print("Diện tích hình CN là:", p.area()) # Gọi PT của ĐT
```

Kết quả:

Diện tích hình CN là: 40

Lập trình HĐT: Lớp-Hàm khởi tạo

- **Hàm khởi tạo** (constructor): PT đặc biệt trong lớp, được gọi tự động khi 1 ĐT mới của lớp được tạo ra. Hàm này được sử dụng để khởi tạo các thuộc tính của ĐT.
- **Cấu trúc hàm khởi tạo trong Python:**

```
def __init__(self, para1, para2, ...):
    self.para1 = para1
    self.para2 = para2 ...
```

 - Tham số bắt buộc **self** đại diện cho ĐT được tạo, được sử dụng để khởi tạo cho các thuộc tính.
- **Cơ chế hoạt động của tham số self:**
 - **self** cho phép truy cập các thuộc tính và PT của ĐT trong phạm vi của lớp.
 - Khi gọi 1 PT của ĐT, Python tự động truyền ĐT vào tham số **self** của PT.

Lập trình HDT: Lớp-Ví dụ

Ví dụ. (Lớp, hàm khởi tạo)

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length ; self.width = width
    def area(self):
        return self.length * self.width
    def perimeter(self):
        return 2 * (self.length + self.width)

my_rectangle = Rectangle(8, 5)
print("Diện tích hình CN:", my_rectangle.area())
```



- ① **self** cho phép các PT *area* và *perimeter* truy cập 2 thuộc tính **length** và **width** của ĐT.
- ② Khi PT *my_rectangle.area()* được gọi, **self** tự động được truyền các tham số qua ĐT **my_rectangle**.

Lập trình HDT: Lớp-Sự kế thừa-1

- **Kế thừa** (inheritance): cho phép tạo lớp mới (**lớp con** - subclass) dựa vào các lớp hiện có (**lớp cha** - superclass). Lớp con này sẽ thừa hưởng các thuộc tính và PT của lớp cha.
- $\leadsto$
 - Một tính năng quan trọng trong LTHDT
 - Cho phép tái sử dụng mã nguồn, giảm thiểu việc lặp lại và tạo cấu trúc phân cấp trong thiết kế lớp.
- **Khai báo lớp con:**
class < tên lớp con > (< tên lớp cha >) :
 < Định nghĩa các *thuộc tính* và *PT* mới cho lớp con >
- **Truy cập PT và thuộc tính của lớp cha:**
 - Lớp con có thể truy cập các PT, thuộc tính của lớp cha:
 - ① Sử dụng hàm ***super()***
 - ② Truy cập trực tiếp.
- **Kiểm tra mối quan hệ kế thừa** (*True, False*):

Lập trình HDT: Lớp-Sự kế thừa-2

- ~→

`class Superclass:`

`def __init__(self, value):`

`self.value = value`

`class Subclass(Superclass):`

`def __init__(self, value, additional_value):`

`super().__init__(value) # Gọi hàm init lớp cha`

`self.additional_value = additional_value`

- ***super()*:**

- 1 Thường được dùng trong PT khởi tạo (`__init__`) của lớp con để gọi PT khởi tạo của lớp cha ⇒ giúp đảm bảo rằng cả lớp con và lớp cha đều được khởi tạo một cách chính xác.
- 2 Cũng có thể được sử dụng để gọi các PT khác của lớp cha bên trong lớp con.

Lập trình HDT: Lớp-Sự kế thừa-3

Ví dụ.

```
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return f"{self.name} sủa gâu gâu!"
class Cat(Animal):
    def speak(self):
        return f"{self.name} kêu meo meo!"

dog = Dog("Bông") ; cat = Cat("Kitty")
print(dog.speak()) ; print(cat.speak())
```

Lập trình HDT: Lớp-Sự kế thừa-4

Ví dụ. (tiếp theo)

Kết quả:

Bông sữa gâu gâu!

Kitty kêu meo meo!

- **Kế thừa nhiều lớp:**
 - Một lớp con có thể kế thừa từ nhiều lớp cha.
`class < tên lớp con > (< tên lớp cha 1 >, < tên lớp cha 2 >, ...):`
`< ... >`
- **Ghi đè PT:**
 - Lớp con có thể ghi đè các PT của lớp cha bằng cách định nghĩa lại chúng trong lớp con.
- ~> Cần thận khi dùng sự kế thừa để tránh việc tạo ra các mối quan hệ phức tạp, khó hiểu trong CT.

Lập trình HĐT: Các thao tác trên lớp-1

- Thay đổi giá trị của các thuộc tính:

$\langle DT \rangle . \langle \text{thuộc tính} \rangle = \langle \text{giá trị mới} \rangle$

- Xóa 1 DT:

del $\langle DT \rangle$

- Gọi 1 PT:

$\langle DT \rangle . \langle PT \rangle$

- So sánh 2 DT kiểu lớp trong Python:

- Các phép toán so sánh: $==, !=, <, <=, >, >=$
- Để sử dụng được các toán tử so sánh này $\Rightarrow$ phải định nghĩa các PT so sánh tương ứng trong lớp. Khi sử dụng các toán tử so sánh, Python sẽ gọi các PT so sánh tương ứng để so sánh 2 DT của lớp.

Lập trình HDT: Các thao tác trên lớp-2

- Các PT tương ứng với các toán tử so sánh:

Toán tử so sánh	PT thực hiện tương ứng
==	<code>__eq__(self, other)</code>
!=	<code>__ne__(self, other)</code>
<	<code>__lt__(self, other)</code>
<=	<code>__le__(self, other)</code>
>	<code>__gt__(self, other)</code>
>=	<code>__ge__(self, other)</code>

- ↪ Các PT này cho phép chúng ta tùy chỉnh hành vi của ĐT của lớp trong các tình huống cụ thể.

Lập trình HDT: Các thao tác trên lớp-3

- Các PT tương ứng với các phép toán số học:

Phép toán số học	PT thực hiện tương ứng
+	<code>__add__(self, other)</code>
-	<code>__sub__(self, other)</code>
*	<code>__mul__(self, other)</code>
/	<code>__truediv__(self, other)</code>
//	<code>__floordiv__(self, other)</code>
%	<code>__mod__(self, other)</code>
**	<code>__pow__(self, other)</code>

Lập trình HĐT: Các thao tác trên lớp-4

- Các PT tương ứng các phép toán khác:

Các phép toán	PT thực hiện tương ứng
<i>len()</i>	<code>__len__(self)</code>
<i>object</i> [key]	<code>__getitem__(self, key)</code>
<i>object</i> [key] = value	<code>__setitem__(self, key, value)</code>
<i>del object</i> [key]	<code>__delitem__(self, key)</code>

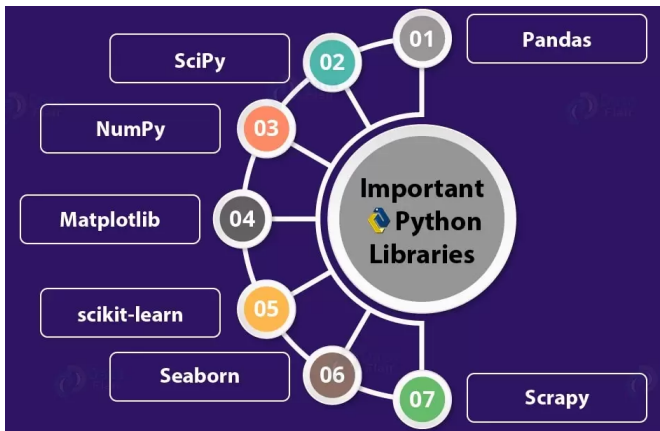
- Một số PT khác:

- ① `__init__(self, ...)`: được gọi khi 1 ĐT mới được tạo ra
- ② `__str__(self)`: trả về 1 chuỗi biểu diễn của ĐT thường được sử dụng khi gọi hàm ***print()***
- ③ `__call__(self, ...)`: cho phép ĐT được gọi như 1 hàm
- ④ `__iter__(self)`: trả về 1 trình lặp cho ĐT
- ⑤ `__next__(self)`: xác định phần tử tiếp theo khi sử dụng trình lặp (***for***) trên ĐT.

Nội dung môn học

- ① Tổng quan về Python
- ② Ngữ nghĩa, cú pháp LT trong Python
- ③ Các kiểu dữ liệu phức tạp của Python
- ④ Lập trình hướng đối tượng trong Python
- ⑤ Một số thư viện chuẩn của Python**
- ⑥ Tài liệu tham khảo

Chương 5. Một số thư viện chuẩn của Python



Các TV mã nguồn mở của Python

Một số TV chuẩn: Thư viện Numpy-1

- Hỗ trợ các phép toán về **số học**, **ma trận**.
- **Những chức năng nổi bật:**
 - **Mảng Numpy** (Numpy array - '*ndarray*'): MNC với tốc độ XL nhanh hơn rất nhiều lần so với các DS thông thường trong Python (*core Python*).
 - **Hàm toán học:** rất mạnh cho XLDL số
 - ① **Số học:** *.add()*, *.subtract()*, *.multiply()*, *.divide()*
 - ② **Lượng giác:** *.sin()*, *.cos()*, *.exp()* ...
 - ③ **TK:** *.mean()*, *.median()*, *.std()* ...
 - **ĐSTT:** hỗ trợ về ma trận
 - **Đọc, ghi DL:** cho các tệp như CSV, TXT hoặc thậm chí của MATLAB
 - **Tích hợp tốt:** Scipy, Matplotlib, Pandas $\Rightarrow$ tạo thành 1 hệ sinh thái mạnh cho tính toán, trực quan hóa và XLDL
 - **Hiệu năng cao:** viết bằng C và được tối ưu hóa

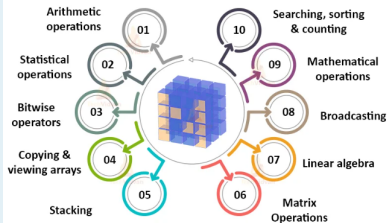
Một số TV chuẩn: Thư viện Numpy-2



- ① **Cài đặt:** *pip install numpy*
- ② **Khuyến khích:** cài Python qua **Anaconda** $\Rightarrow$ có kèm theo *Numpy, Pandas, Matplotlib, Scipy ...*



Uses of NumPy

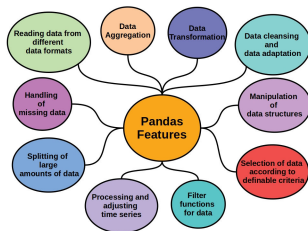
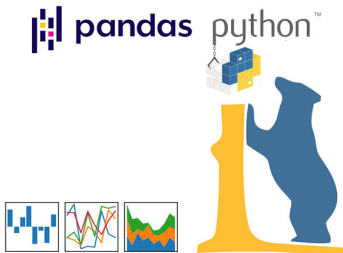


Một số TV chuẩn: Thư viện Pandas-1

- Hỗ trợ cho PTDL, XLDL dạng bảng (**DataFrame**, **Series**).
- Một số đặc trưng quan trọng:
 - **DataFrame**: có dạng bảng với dòng và cột ($\Leftrightarrow$ bảng trong *SQL*, *Excel*), cho phép lưu trữ và XLDL 2 chiều 1 cách dễ dàng
 - **Series**: có dạng M1C ($\Leftrightarrow$ 1 cột, hàng trong DataFrame)
 - **XLDL**:
 - Có thể nạp DL từ nhiều nguồn: *CSV*, *Excel*, *SQL* ...
 - Nhiều thao tác XLDL: *subsetting*, *slicing*, *filtering*, *merging*, *groupBy* ...
 - XLDL thiếu: dễ dàng
 - **Tích hợp tốt**: Numpy, Matplotlib
 - **Truy vấn DL** ($\Leftrightarrow$ *SQL*)
 - **PTDL** ($\Leftrightarrow$ mô hình hoá và TK)
- **Hiệu suất rất tốt**: thậm chí hơn cả sử dụng **Cython**

Một số TV chuẩn: Thư viện Pandas-2

- **Lợi ích của Pandas** (so với sử dụng Java, C, C++):
 - ① **Biểu diễn DL:** một cách tự nhiên và dễ dàng, trong khi dùng với các NN khác sẽ mất nhiều công code hơn
 - ② **Tập hợp và lọc DL:** có nhiều PT cho việc PTDL
 - ③ **Code:** ngắn gọn và rõ ràng hơn, cho phép NSD tập trung vào những mục tiêu chính hơn là việc phải viết 1 đoạn code dài từ đầu để thực hiện công việc
- **Cài đặt:** *pip install pandas*



Một số TV chuẩn: Thư viện Scipy-1

- Hỗ trợ cho KHDL, tính toán KH và NC. Được xây dựng dựa trên *Numpy* (kiến thức về toán).
- **Một số đặc trưng quan trọng:**
 - **Toán học, KH:** nhiều mô-đun con như *tối ưu hóa*, *PTVP*, *tích phân*, *ĐSTT*, *TK*, *LTXS* ...
 - **Tối ưu hóa:** nhiều *TT* tối ưu
 - **TK:** nhiều công cụ TK mạnh (*kiểm định giả thuyết*, *phân phối XS*, *tính toán độ tương quan* ...)
 - **Xử lý tín hiệu và hình ảnh:** *biến đổi Fourier*, *lọc tín hiệu*, *xử lý hình ảnh*
- **Hiệu năng:** rất mạnh được nhiều nhà KH và kỹ sư hàng đầu thế giới lựa chọn, dễ sử dụng



Một số TV chuẩn: Thư viện Skipy-2

- Chứa hơn 15 gói con cho ĐSTT, tối ưu hóa, phân cụm DL, XL hình ảnh và tín hiệu, TK ...
 - 1 *linalg*: ĐSTT
 - 2 *optimize*: Tối ưu hóa
 - 3 *integrate*: PTVP và tích phân
 - 4 *fftpack*: Biến đổi Fourier nhanh
 - 5 *interpolate*: Nội suy
 - 6 *odr*: Hồi quy
 - 7 *special*: Các hàm toán học đặc biệt
 - 8 *stats*: Các hàm và phân phối TK
 - 9 *ndimage*: Xử lý ảnh nhiều chiều
 - 10 *signal*: Xử lý tín hiệu
 - 11 *cluster*: TT phân cụm
 - 12 ...

Một số TV chuẩn: Thư viện Matplotlib-1

- Vẽ biểu đồ và trực quan hóa DL.
- Một số đặc trưng quan trọng:
 - Đa dạng hóa biểu đồ (2D, 3D): *biểu đồ đường, cột, phân tán, hình tròn, đám mây ...*
 - Tùy biến cao: có thể tùy chỉnh hầu hết mọi khía cạnh của biểu đồ: *tiêu đề, trục, màu sắc, kích thước ...*
 - Tương thích nhiều DL: *mảng (Numpy), DataFrame (Pandas) ...*
 - Lưu trữ, chia sẻ: *tệp ảnh, véc-tơ (PDF, PNG ...)*
 - Tương thích cao: IPython, Jupyter Notebook
 - Nhiều gói mở rộng: **seaborn, Plotly**
 - TV con Pyplot: tạo và tùy chỉnh biểu đồ một cách dễ dàng, cung cấp giao diện như **MATLAB**



Một số TV chuẩn: Thư viện Scikit learn (sklearn)

- Chuyên về HM, KHDL. Được xây dựng trên nền tảng của Numpy, Scipy và Matplotlib.
- **Một số đặc trưng quan trọng:**
 - **HM:** các TTHM cơ bản như *HQTT, CQĐ, rừng NN, SVM, k-means clustering, giảm chiều ...*
 - **XL DL:** chuẩn hóa, mã hóa, XL DL thiếu, biến đổi DL, chia DL thành THL và TKT ...
 - **Đánh giá MHHM:** độ chính xác, độ đo F1 ...
 - **Lựa chọn đặc trưng:** tự động
 - **Phân tích, trực quan hóa DL:** *t-SNE, PCA*
 - **XL văn bản, DL dạng chuỗi:** *vectơ hóa Tfidf, Word Embeddings*



Một số TV chuẩn: Thư viện Seaborn

- Trực quan hóa những DL TK phức tạp 1 cách dễ dàng, hấp dẫn và đẹp. Được xây dựng dựa trên Matplotlib, và rất dễ sử dụng.
- **Một số đặc trưng quan trọng:**
 - **Trực quan hóa:** rất mạnh để tạo biểu đồ TK như *biểu đồ hộp, Violin, đám mây, phân phối, ROC, phân loại, biểu đồ đánh giá hiệu suất MH ...*
 - **Tích hợp tốt:** Pandas
 - **Tùy biến cao:** màu sắc, phong cách, giao diện ...



Một số TV chuẩn: Thư viện Scrapy

- Trích xuất DL từ trang web một cách tự động, siêu mạnh.
- DL sau khi bóc tách sẽ được lưu trữ dưới dạng DL có cấu trúc: *CSV*, *JSON*.



Nội dung môn học

- ① Tổng quan về Python
- ② Ngữ nghĩa, cú pháp LT trong Python
- ③ Các kiểu dữ liệu phức tạp của Python
- ④ Lập trình hướng đối tượng trong Python
- ⑤ Một số thư viện chuẩn của Python
- ⑥ Tài liệu tham khảo**

Tài liệu tham khảo

- ① Dave Kuhlman, *A Python Book: Beginning Python, Advanced Python and Python Exercises*, 2013.
- ② Nguyễn Văn Hậu, *Python cơ bản*, NXB Đại học Quốc Gia Hà nội, 2019.
- ③ Bùi Việt Hà, *Python cơ bản*, NXB Đại học Quốc Gia Hà nội, 2021.



Bảng viết tắt

Viết tắt	Ý nghĩa
PT	Phương thức
DL	Dữ liệu
KDL	Kiểu dữ liệu
BC	Biến chuỗi
BCC	Biến chuỗi con
DS	Danh sách
TĐ	Từ điển
NLT	Người lập trình
ĐQ	Độ quy
PP	Phương pháp
BDS	Biến danh sách
BTĐ	Biến từ điển
BTH	Biến tập hợp

Bảng viết tắt

Viết tắt	Ý nghĩa
BB	Biến bộ
BT	Biến tệp
ĐT	Đối tượng
HĐT	Hướng đối tượng
HTT	Hướng thủ tục
pt	Phần tử
BK	Biến khóa
BGT	Biến giá trị
TB	Tên biến
DLTK	Dữ liệu thống kê
LTHĐT	LT hướng đối tượng
MNC	Mảng nhiều chiều

Phụ lục

- ❶ Trả về thư mục hiện tại (current working directory):

```
import os # operating system  
print(os.getcwd())
```

- ❷ Lấy ra đường dẫn đến thư mục my_module ở trong project hiện hành:

```
import os, sys  
lib_path = os.path.abspath(os.path.join('modules'))
```

- ❸ Thêm thư mục cần load vào trong hệ thống:

```
sys.path.append(lib_path)
```

Phu lục

