

LẬP TRÌNH VỚI PYTHON

(Programming with Python)

Th.S Nguyễn Hoàng Thành

Email: thanhnh@ptithcm.edu.vn

Tel: 0909 682 711

1. KHÁI NIỆM

1.1 Từ khóa (keyword) trong Python

- Từ khóa (keyword) là những từ (word) được dành riêng trong Python.
- Chúng ta **KHÔNG** thể sử dụng từ khóa để đặt tên biến, tên hàm hoặc bất kỳ định danh (**identifier**) nào khác.
- Chúng được sử dụng để xác định cú pháp và cấu trúc của ngôn ngữ Python. Trong Python, các từ khóa có sự phân biệt chữ hoa và chữ thường.

1.1 Từ khóa (keyword) trong Python (tt)

- Để xem các từ khóa trong phiên bản Python dùng lệnh

`help("keywords")`

```
>>> help("keywords")
```

Here is a list of the Python keywords. Enter any keyword to get more help.

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

Các từ khóa trong Python 3.7

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

1.2 Định danh (identifier) trong Python

- ❖ Định danh (identifier) là tên được đặt cho các **thực thể như lớp, hàm, biến,...**
- ❖ Định danh giúp **phân biệt thực thể** này với thực thể khác.

1.2 Định danh (identifier) trong Python (tt)

❖ **Những quy tắc** khi đặt tên định danh

1. Tên định danh có thể bao gồm các chữ thường (a đến z), chữ hoa (A đến Z), chữ số (0 đến 9), dấu gạch dưới `_`. Tên định danh không được bắt đầu bằng một chữ số. Ví dụ, tên định danh `1variable` không hợp lệ nhưng `variable1` thì hợp lệ.
2. Không được đặt tên định danh giống với từ khóa (keyword).
3. Không được sử dụng các ký hiệu đặc biệt như `!`, `@`, `#`, `$`, `%`, ... trong tên định danh.
4. Tên định danh có thể có độ dài bất kỳ.

1.2 Định danh (identifier) trong Python

❖ Một số lưu ý cần nhớ

- Python **phân biệt chữ hoa và chữ thường**. Do đó, tên định danh VaRiable và variable là khác nhau.
- Nên đặt tên định danh có **ý nghĩa** và **dễ nhớ**. Thay vì đặt tên định danh của biến là `c = 10` thì có thể đặt là `count = 10`. Lúc này, tên định danh của biến sẽ rõ nghĩa hơn và cho biết biến `count` là một biến dùng để lưu một giá trị đếm.

1.2 Định danh (identifier) trong Python

❖ Một số lưu ý cần nhớ

- Các từ trong một tên định danh có thể được **nối với nhau bởi dấu gạch dưới**. Ví dụ như `this_is_a_long_variable`.
- Tên định danh **không bao gồm ký tự khoảng trắng**. Ví dụ, tên định danh `count a` là không hợp lệ nhưng `counta` thì hợp lệ.

Ví dụ về các định danh hợp lệ:

```
>>> a = 1
```

```
>>> _a = 2
```

```
>>> a1 = 3
```

```
>>> aA2_ = 4
```

Ví dụ về các định danh hợp lệ:

- Khai báo biến không hợp lệ do tên biến bắt đầu bằng 1 số. Và nhận báo lỗi **Syntax Error**

```
>>> 1a = 2
      File "<stdin>", line 1
        1a = 2
          ^
SyntaxError: invalid syntax
>>>
```

Ví dụ về các định danh hợp lệ:

- Khai báo biến không hợp lệ do tên chứa khoảng trắng. Và nhận báo lỗi **Syntax Error**

```
>>> a b = 2
      File "<stdin>", line 1
        a b = 2
          ^
```

```
SyntaxError: invalid syntax
```

```
>>>
```

Ví dụ về các định danh hợp lệ:

- Khai báo biến không hợp lệ do tên chứa ký tự đặc biệt. Và nhận báo lỗi **Syntax Error**

```
>>> a_* = 3
      File "<stdin>", line 1
        a_* = 3
            ^
SyntaxError: invalid syntax
>>>
```

1.3 Câu lệnh (statement) trong Python

- Python sẽ thông dịch từng câu lệnh (statement) để thực thi. Một statement trong Python thường được viết trong 1 dòng. Chúng ta không cần thiết phải thêm dấu chấm phẩy ; vào cuối mỗi câu lệnh. Ví dụ:

```
>>> a = 5
>>> b = 10
>>> print("Tong =", a + b)
Tong = 15
```

- Chúng ta cũng có thể **đặt nhiều câu lệnh** trong một dòng bằng cách sử dụng dấu chấm phẩy ; như sau:

```
>>> a = 1; b = 2; c = 3
```

1.3 Câu lệnh (statement) trong Python

- Chúng ta có thể viết một câu lệnh trên nhiều dòng bằng cách sử dụng thích hợp các ký tự như ký tự tiếp tục dòng (\), dấu ngoặc đơn (), ngoặc vuông [], ngoặc nhọn {}.

```
>>> a = 1 + 2 + 3 + \
...     4 + 5 + 6 + \
...     7 + 8 + 9
>>> a = (1 + 2 + 3 +
...     4 + 5 + 6 +
...     7 + 8 + 9)
>>> colors = ['red',
...            'blue',
...            'green']
```

Tiếp tục dòng khi dùng \, (), []

1.4 Thụt đầu dòng (indentation) trong Python

- Python sử dụng thụt đầu dòng (indentation) để định nghĩa một khối lệnh (code block) như thân hàm, thân vòng lặp,...
- Lưu ý: Python không sử dụng dấu ngoặc nhọn {} cho code block như các ngôn ngữ C/C++, Java,...

```
>>> for i in range(1,11):
```

```
...     print(i)
```

```
...     if i == 5:
```

```
...         break
```

```
...
```

```
1
```

```
2
```

```
3
```

```
4
```

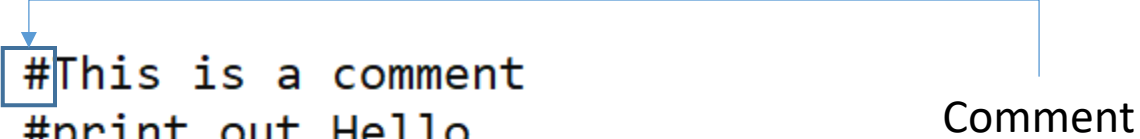
```
5
```

Các thụt dòng có
khoảng cách như nhau

1.5. Ghi chú (comment) trong Python

- **Ghi chú (comment)** được sử dụng để giải thích code đang thực hiện những gì.
- Việc này rất quan trọng khi đọc lại source code, bảo trì chương trình sau này. Sử dụng ký hiệu hash (#) để bắt đầu viết comment trong Python.
- Trình thông dịch Python sẽ **bỏ qua comment** bắt đầu từ ký hiệu hash (#) cho đến khi gặp ký tự bắt đầu dòng mới.

```
>>> #This is a comment
... #print out Hello
... print('Hello')
Hello
>>>
```



Comment

1.5. Ghi chú (comment) trong Python

- Chúng ta có thể viết **comment trên nhiều dòng**, mỗi dòng bắt đầu bằng ký tự hash (#). Hoặc một cách khác để comment trên nhiều dòng bằng cách sử dụng dấu nháy ' hoặc """. Ví dụ:

```
>>> #This is a comment
... #print out Hello
... print('Hello')
Hello
>>> """This is also a
... perfect example of
... multi-line comments"""
'This is also a\nperfect example of\nmulti-line comments'
```

1.6. Khối lệnh (code block) trong Python

- Một hoặc nhiều câu lệnh (**statement**) có thể tạo thành một khối lệnh (**code block**).
- Các khối lệnh thường gặp trong Python như các lệnh trong **lớp (class), hàm (function), vòng lặp (loop),...**
- Python sử dụng **thụt đầu dòng (indentation)** để bắt đầu định nghĩa, phân tách một code block với các code block khác.

1.6. Khối lệnh (code block) trong Python

```
>>> import sys
>>> file = "data.txt"
>>> try:
...     myfile = open(file, "r")
...     myline = myfile.readline()
...     while myline:
...         print(myline)
...         myline = myfile.readline()
...     myfile.close()
... except IOError as e:
...     print("I/O error")
... except:
...     print("Unexpected errpr")
... 
```

**Khối lệnh
thân while**



1.6. Khối lệnh (code block) trong Python

```
>>> import sys
>>> file = "data.txt"
>>> try:
...     myfile = open(file, "r")
...     myline = myfile.readline()
...     while myline:
...         print(myline)
...         myline = myfile.readline()
...     myfile.close()
... except IOError as e:
...     print("I/O error")
... except:
...     print("Unexpected errpr")
... 
```

**Khối lệnh
thân try**



1.6. Khối lệnh (code block) trong Python

```
>>> import sys
>>> file = "data.txt"
>>> try:
...     myfile = open(file, "r")
...     myline = myfile.readline()
...     while myline:
...         print(myline)
...         myline = myfile.readline()
...     myfile.close()
... except IOError as e:
...     print("I/O error")
... except:
...     print("Unexpected errpr")
... 
```

**Khối lệnh
thân except**



1.7. Docstring trong Python

- **Docstring** là viết tắt của từ documentation string, tạm dịch là chuỗi tài liệu trong Python.
- Docstring là chuỗi ký tự xuất hiện ngay sau khi định nghĩa của một phương thức, lớp hoặc module.
- Sử dụng dấu nháy `"""` để viết docstring.
- Chúng ta có thể truy cập docstring với thuộc tính `__doc__`.

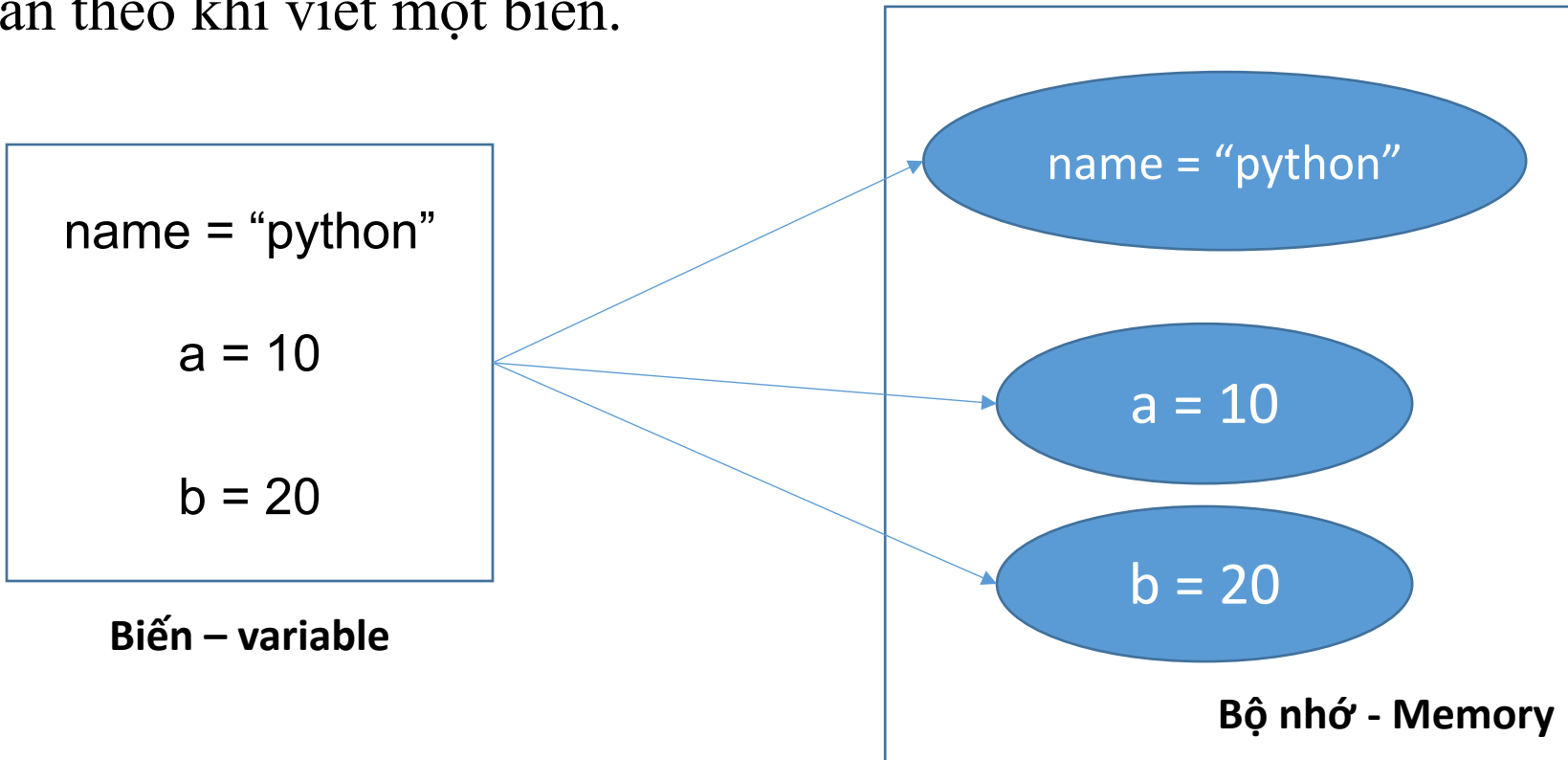
```
>>> def double(num):  
...     """Function to double the value"""  
...     return 2*num  
...  
>>> print(double.__doc__)  
Function to double the value
```

Docstring

Truy cập Docstring

1.8 Biến – Variable

- Một biến Python được tạo ngay sau khi một giá trị được gán cho nó. Nó không cần bất kỳ lệnh bổ sung nào để khai báo một biến trong python.
- Có một số quy tắc và quy định nhất định mà chúng ta phải tuân theo khi viết một biến.



Định nghĩa và khai báo biến

- Python không có lệnh bổ sung để khai báo một biến. Ngay sau khi giá trị được gán cho nó, biến được khai báo.

```
>>>
```

```
>>> name = "python"
```

```
>>> a = 10
```

```
>>> b = 3
```

```
>>>
```

Giá trị – value của biến name
là **chuỗi "python"**

Biến – Variable tên là **name**

Định nghĩa và khai báo biến (tt)

- Python không có lệnh bổ sung để khai báo một biến. Ngay sau khi giá trị được gán cho nó, biến được khai báo.

```
>>>
```

```
>>> name = "python"
```

```
>>> a = 10
```

```
>>> b = 3
```

```
>>>
```

Giá trị – value của biến **a** là **số 10**

Biến – Variable tên là **a**

Định nghĩa và khai báo biến (tt)

- Python không có lệnh bổ sung để khai báo một biến. Ngay sau khi giá trị được gán cho nó, biến được khai báo.

```
>>>
```

```
>>> name = "python"
```

```
>>> a = 10
```

```
>>> b = 3
```

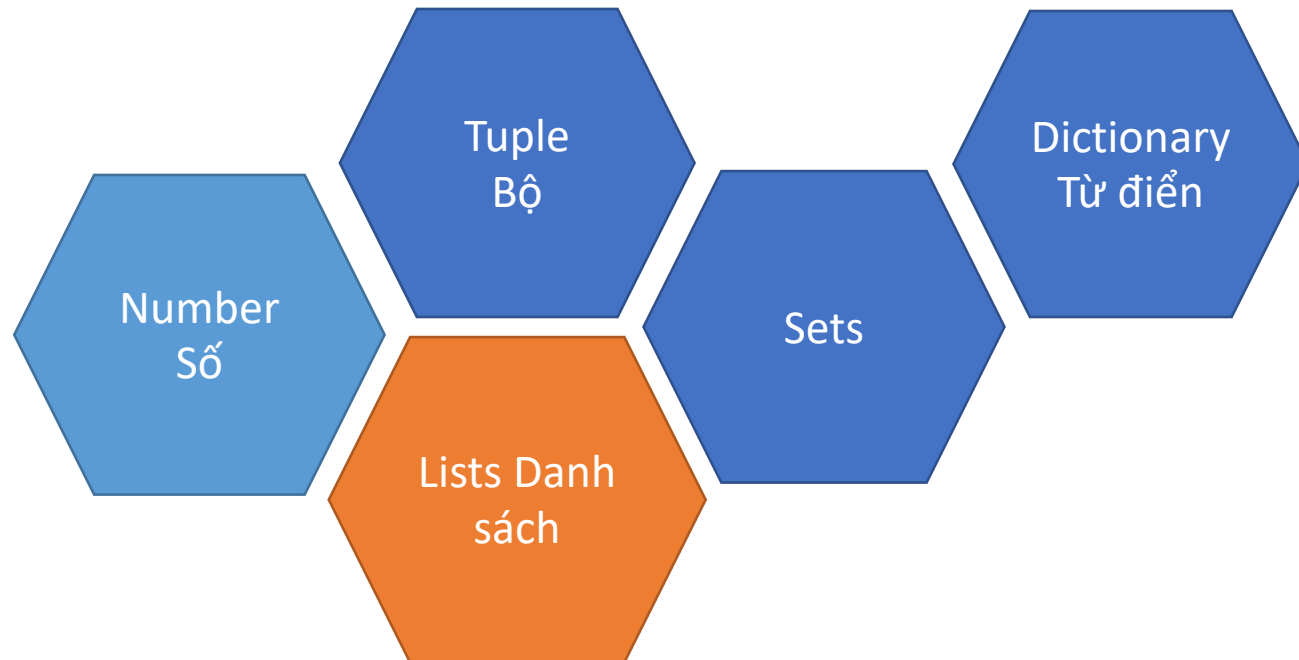
```
>>>
```

Giá trị – value của biến **b** là **số 3**

Biến – Variable tên là **b**

1.9 Kiểu dữ liệu (Data type)

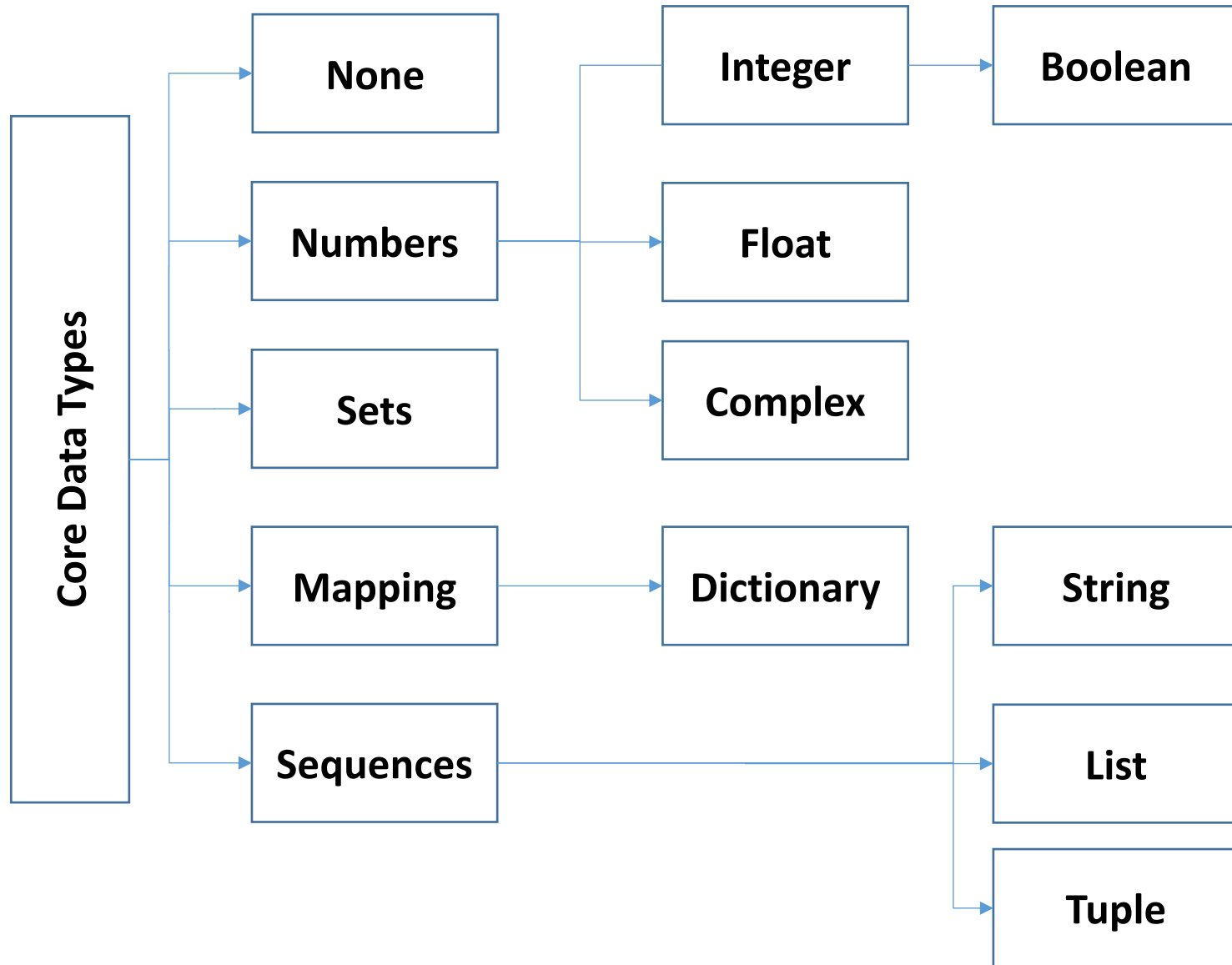
- Có một số loại dữ liệu trong python.
- Mọi giá trị mà chúng ta khai báo (declare) trong python đều có một kiểu dữ liệu.
- Các **kiểu dữ liệu** là các **lớp (class)** và các **biến (variable)** là các **thể hiện (instances)** của các lớp này.



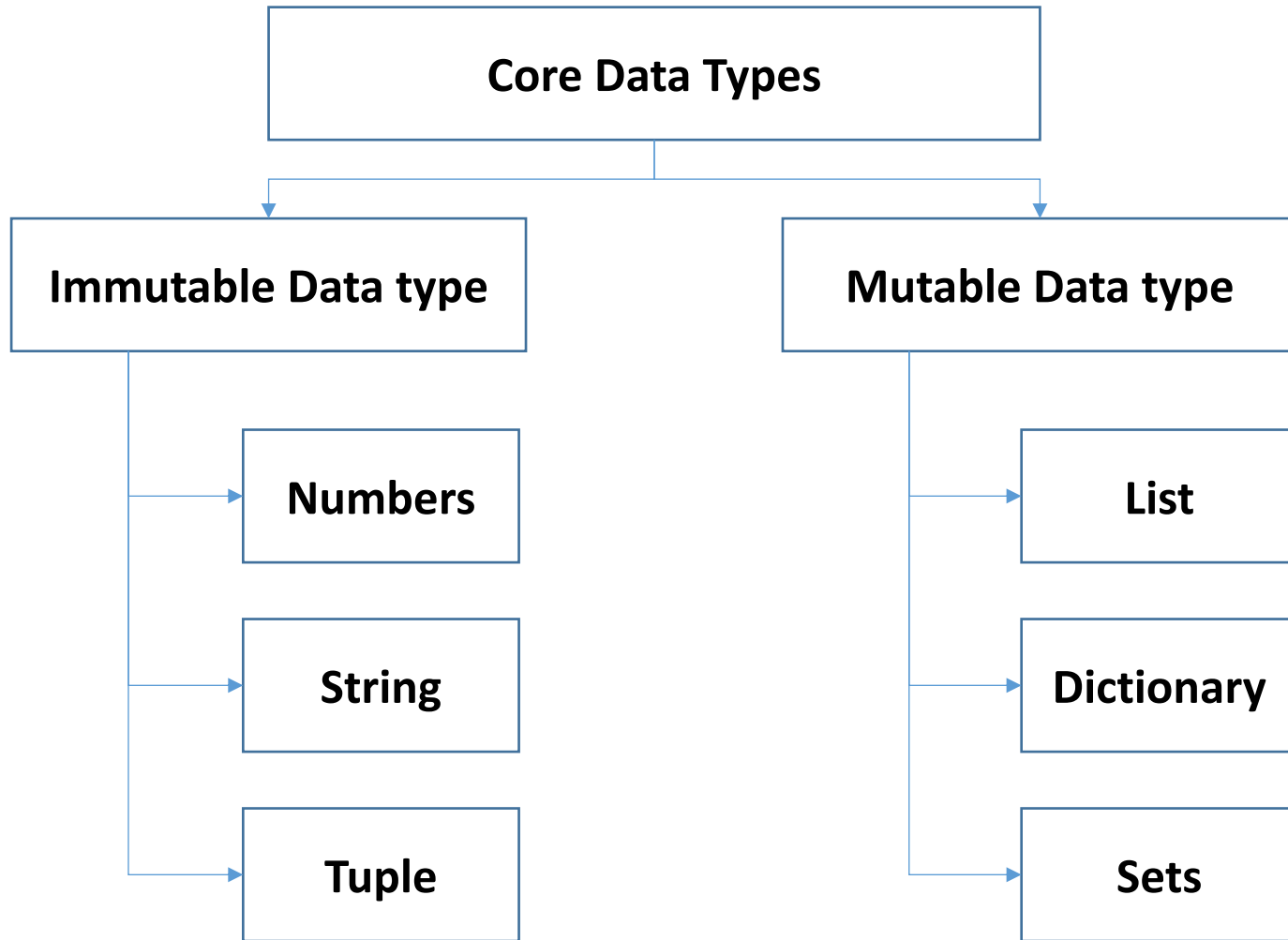
Python's Core Data Types

Object type	Example literals/creation
Numbers	<code>1234, 3.1415, 3+4j, Decimal, Fraction</code>
Strings	<code>'spam', "guido's", b'a\x01c'</code>
Lists	<code>[1, [2, 'three'], 4]</code>
Dictionaries	<code>{'food': 'spam', 'taste': 'yum'}</code>
Tuples	<code>(1, 'spam', 4, 'U')</code>
Files	<code>myfile = open('eggs', 'r')</code>
Sets	<code>set('abc'), {'a', 'b', 'c'}</code>
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes (Part IV , Part V , Part VI)
Implementation-related types	Compiled code, stack tracebacks (Part IV , Part VII)

Python's Core Data Types (con't)



Python's Core Data Types (con't)



1.9.1 Các kiểu dữ liệu số trong Python

- Kiểu dữ liệu số chứa giá trị số. Trong dữ liệu số cũng có 4 loại phụ. Sau đây là các kiểu con của kiểu dữ liệu số:
 - Integers (Số nguyên)
 - Float (Số thực)
 - Complex Numbers (Số phức)
 - Boolean (Logic)

1.9.1 Các kiểu dữ liệu số trong Python (tt)

- **Số nguyên** được sử dụng để đại diện cho các giá trị số nguyên.
- Để kiểm tra kiểu của bất kỳ kiểu dữ liệu biến nào, chúng ta có thể sử dụng hàm ***type()***. Nó sẽ trả về kiểu của kiểu dữ liệu biến được đề cập.

```
>>>  
>>> x = 100  
>>> y = 200  
>>> type(x), type(y)  
(<class 'int'>, <class 'int'>)  
>>>
```

1.9.1 Các kiểu dữ liệu số trong Python (tt)

- Kiểu dữ liệu **float** được sử dụng để biểu thị các giá trị dấu thập phân.
- Để kiểm tra kiểu của bất kỳ kiểu dữ liệu biến nào, chúng ta có thể sử dụng hàm **type()**. Nó sẽ trả về kiểu của kiểu dữ liệu biến được đề cập.

```
>>> x = 10.25
>>> y = 5.15
>>> type(x), type(y)
(<class 'float'>, <class 'float'>)
>>>
```

1.9.1 Các kiểu dữ liệu số trong Python (tt)

- **Số phức - complex** được dùng để biểu diễn các giá trị ảo. Các giá trị ảo được biểu thị bằng **'j'** ở cuối số.

```
>>>  
>>> x = 5 + 10j  
>>> y = 10 + 5j  
>>> type(x), type(y)  
(<class 'complex'>, <class 'complex'>)  
>>>
```

1.9.1 Các kiểu dữ liệu số trong Python (tt)

- **Boolean** được sử dụng cho đầu ra phân loại, vì đầu ra của boolean là **đúng** hoặc **sai**.

```
>>>
>>> num = 10 < 3
>>> type(num)
<class 'bool'>
>>> print(num)
False
>>>
```

num là biến boolean

Kiểu dữ liệu là **bool**

Giá trị biến num là **False** vì **10 < 3** là **False**

1.9.2 Chuỗi

- Các chuỗi trong python được sử dụng để biểu thị các giá trị ký tự **unicode**. Python **không có kiểu dữ liệu ký tự**, một ký tự đơn lẻ cũng được coi là một chuỗi.
- Có thể khai báo các giá trị chuỗi bên trong dấu ngoặc đơn hoặc dấu ngoặc kép. Để truy cập các giá trị trong một chuỗi, chúng ta sử dụng các **chỉ mục (index)** và **dấu ngoặc vuông**.

```
>>> ngonNgu = "Python"
>>> ngonNgu = 'Python'
>>> ngonNgu[0]
'P'
>>> ngonNgu[3]
'h'
>>> ngonNgu[5]
'n'
```

index là 0 và giá trị trả về là **P**

1.9.2 Chuỗi

- Các chuỗi trong python được sử dụng để biểu thị các giá trị ký tự **unicode**. Python **không có kiểu dữ liệu ký tự**, một ký tự đơn lẻ cũng được coi là một chuỗi.
- Có thể khai báo các giá trị chuỗi bên trong dấu ngoặc đơn hoặc dấu ngoặc kép. Để truy cập các giá trị trong một chuỗi, chúng ta sử dụng các **chỉ mục (index)** và **dấu ngoặc vuông**.

```
>>> ngonNgu = "Python"
>>> ngonNgu = 'Python'
>>> ngonNgu[0]
'p'
>>> ngonNgu[3]
'h'
>>> ngonNgu[5]
'n'
```

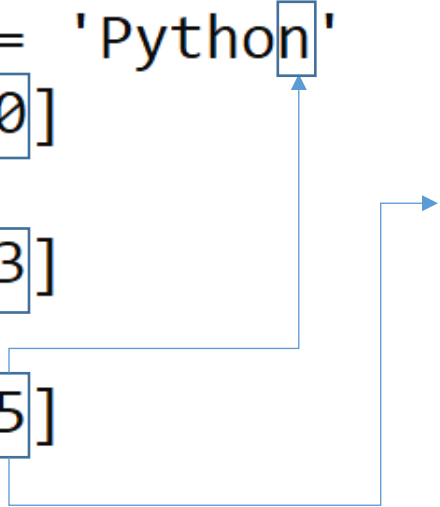
index là 3 và giá trị trả về là 'h'

1.9.2.1 Các khái niệm

- Các chuỗi trong python được sử dụng để biểu thị các giá trị ký tự **unicode**. Python **không có kiểu dữ liệu ký tự**, một ký tự đơn lẻ cũng được coi là một chuỗi.
- Có thể khai báo các giá trị chuỗi bên trong dấu ngoặc đơn hoặc dấu ngoặc kép. Để truy cập các giá trị trong một chuỗi, chúng ta sử dụng các **chỉ mục (index)** và **dấu ngoặc vuông**.

```
>>> ngonNgu = "Python"
>>> ngonNgu = 'Python'
>>> ngonNgu[0]
'p'
>>> ngonNgu[3]
'h'
>>> ngonNgu[5]
'n'
```

index là 5 và giá trị trả về là **'n'**



1.9.2.1 Các khái niệm (tt)

Chuỗi là bất biến (immutable)

- Các chuỗi có bản chất là **bất biến (immutable)**, có nghĩa là **KHÔNG THỂ THAY ĐỔI MỘT CHUỖI** sau khi đã thay thế.

```
>>> s1 = 'SLICEOFSPAM'
```

```
>>> s1[0] = 5
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: 'str' object does not support  
item assignment
```


1.9.2.2 Các toán tử với chuỗi

- Toán tử +: Thực hiện việc nối các chuỗi lại với nhau.
- Cú pháp: **s1 + s2** (với s1 và s2 đều là chuỗi)

```
>>> s1 = "Hello"  
>>> s2 = "Python"  
>>> s1 + s2  
'HelloPython'
```

1.9.2.2 Các toán tử với chuỗi (tt)

- Toán tử *: Thực hiện việc tạo ra một chuỗi lặp đi lặp lại.
- Cú pháp: $s * n$
 - Trong đó:
 - s là biến chuỗi hoặc một chuỗi
 - n là số nguyên chỉ số lần lặp của s .

```
>>> s = "Hello"
>>> s * 4
'HelloHelloHelloHello'
>>> "Python"*3
'PythonPythonPython'
```

1.9.2.2 Các toán tử với chuỗi (tt)

- Toán tử *in*: Kiểm tra một chuỗi con có nằm trong một chuỗi mẹ hay không? Kết quả trả về True nếu có hoặc False nếu không.
- Cú pháp: **s1 in s** với s1 và s là chuỗi

```
>>> s1 = "ello"
>>> s2 = "Hello"
>>> s1 in s2
True
>>> "ello" in "Hello"
True
>>> "Kello" in "Hello"
False
```

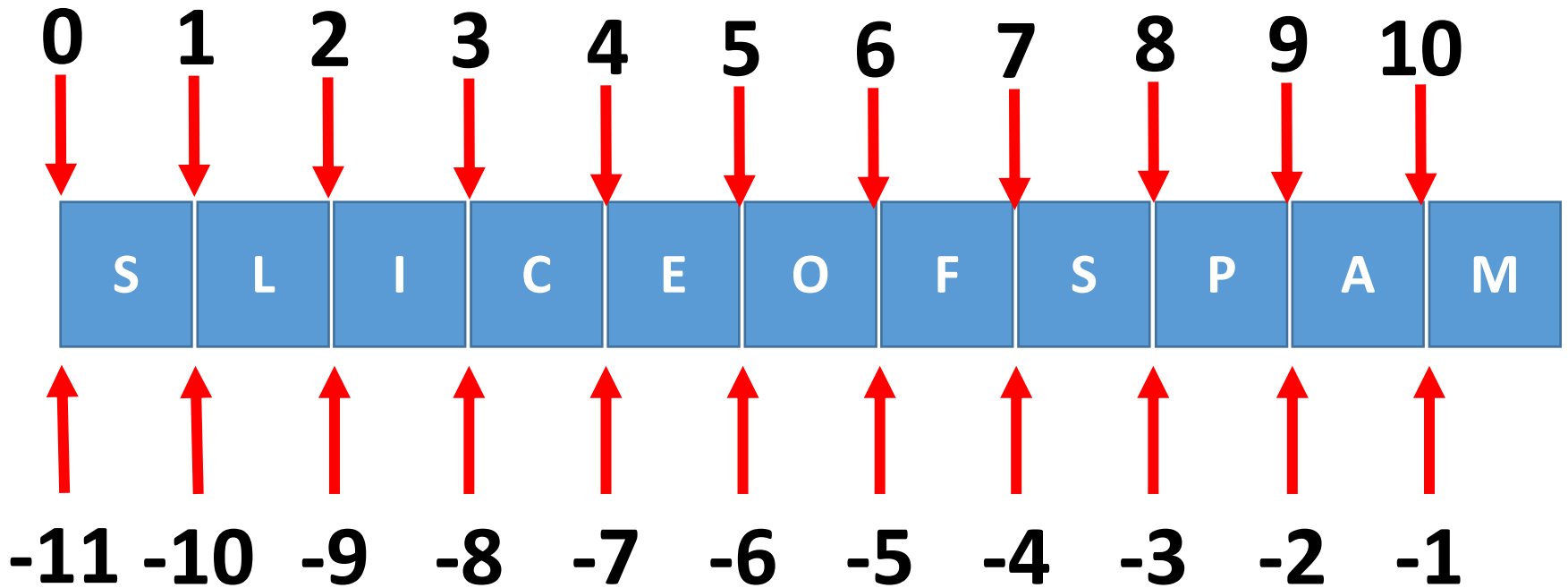
1.9.2.3 Truy cập chuỗi

- Vị trí các ký tự trong chuỗi: Trong một chuỗi của Python,
 - các ký tự bên trái trong chuỗi được đánh số thứ tự từ **0** đến **$n-1$** từ **trái qua phải**. Các số thứ tự này được gọi là index dương.
 - Các ký tự bên phải trong chuỗi được đánh số thứ tự từ **-1** đến **$-n$** từ **phải qua trái**. Các số thứ tự này được gọi là index âm.

Index trong Python

- Index là **vị trí** của một phần tử trong **iterable** (list, chuỗi..), có hai kiểu là **index dương** và **index âm**. Được đánh số từ **0** cho tới **n-1**, từ **trái qua phải**, với n là số ký tự có trong chuỗi.
- Mỗi phần tử đều được đại diện bằng một cặp **index dương** và **âm** để thể hiện vị trí của nó. Và **khoảng cách** cũng được tính là một index.

1.9.2.3 Truy cập chuỗi (tt)



1.9.2.3 Truy cập chuỗi (tt)

❖ Truy xuất đến một phần tử trong chuỗi: Từ một biến kiểu chuỗi cho trước, ta có thể truy xuất đến bất cứ ký tự nào có trong chuỗi.

❖ Cú pháp:

<Tên biến chuỗi>[<vị trí - index>]

➤ Trong đó:

<Tên biến chuỗi>: Là tên biến kiểu chuỗi hoặc một chuỗi, bắt buộc phải có.

<vị trí - index>: Là một số nguyên chỉ vị trí của ký tự trong chuỗi.

1.9.2.3 Truy cập chuỗi (tt)

❖ **Truy xuất đến nhiều phần tử trong chuỗi:** Dựa trên vị trí của các ký tự trong chuỗi, ta có thể lấy ra nhiều ký tự có trong chuỗi.

❖ Cú pháp:

<Tên biến chuỗi>[<start>:<end>:<step>]

➤ Trong đó:

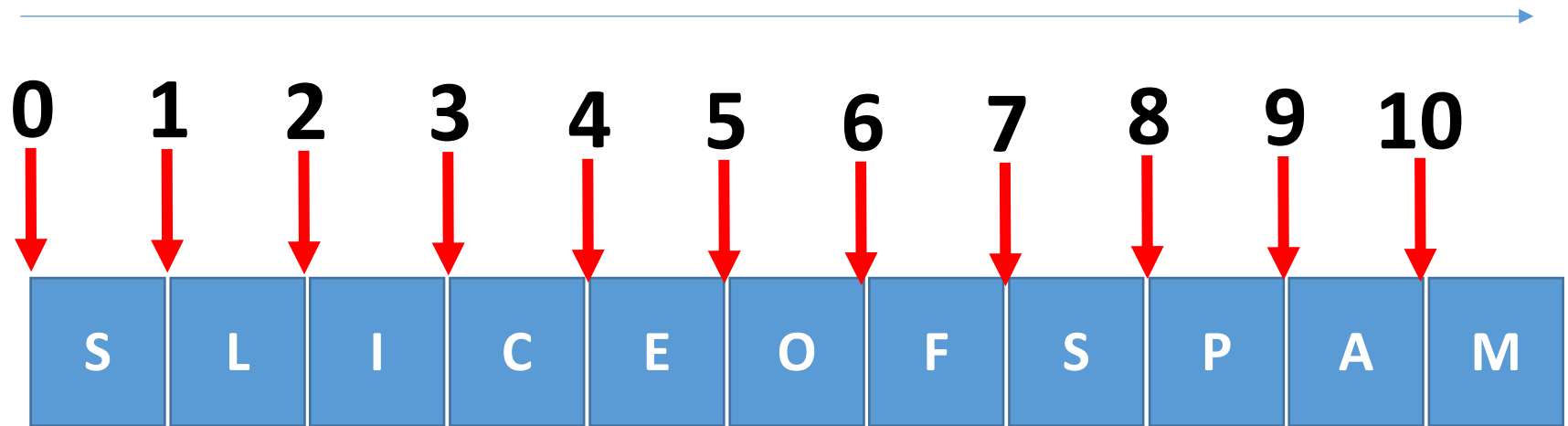
<Tên biến chuỗi>: Là tên biến kiểu chuỗi hoặc một chuỗi, bắt buộc phải có.

<start>: Là một số nguyên chỉ vị trí bắt đầu lấy trong chuỗi, nếu không thì python mặc định lấy giá trị là 0.

<end>: Là một số nguyên chỉ vị trí kết thúc lấy trong chuỗi bằng $end - 1$, nếu không có thì python mặc định lấy đến ký tự cuối cùng của chuỗi.

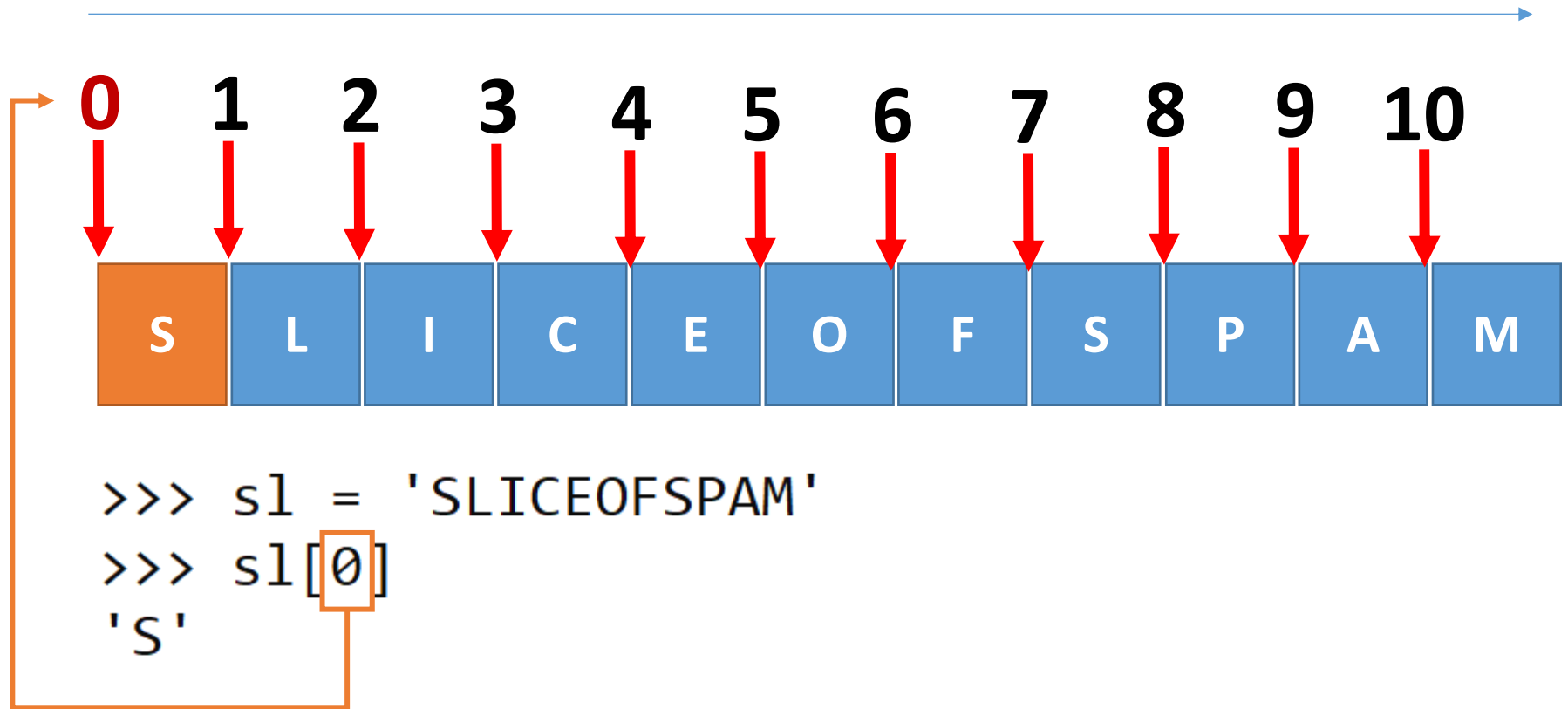
<step>: Là số bước nhảy sẽ lấy, mặc định là 1.

Truy cập chuỗi với Index dương

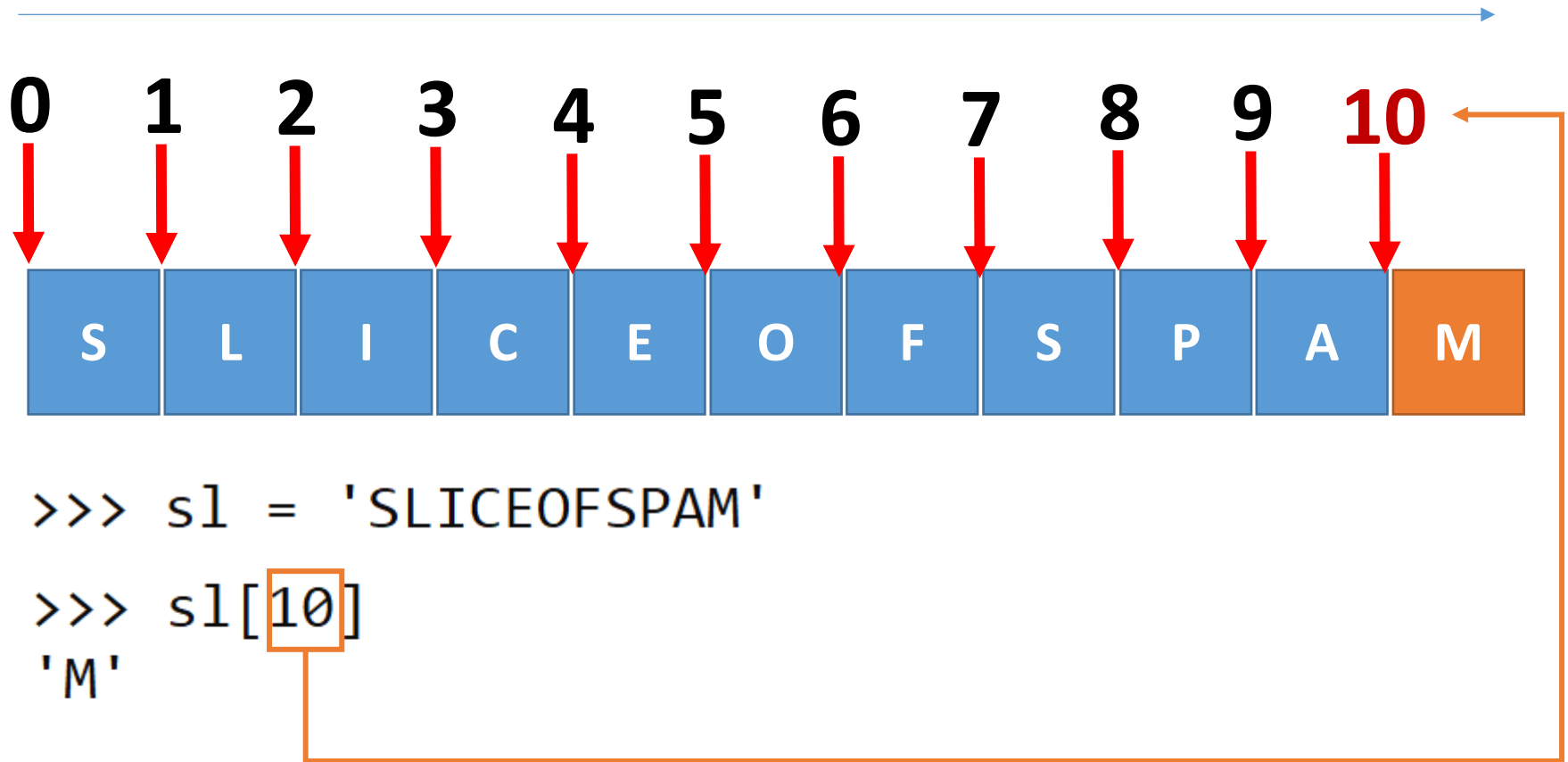


```
>>> s1 = 'SLICEOFSPAM'
```

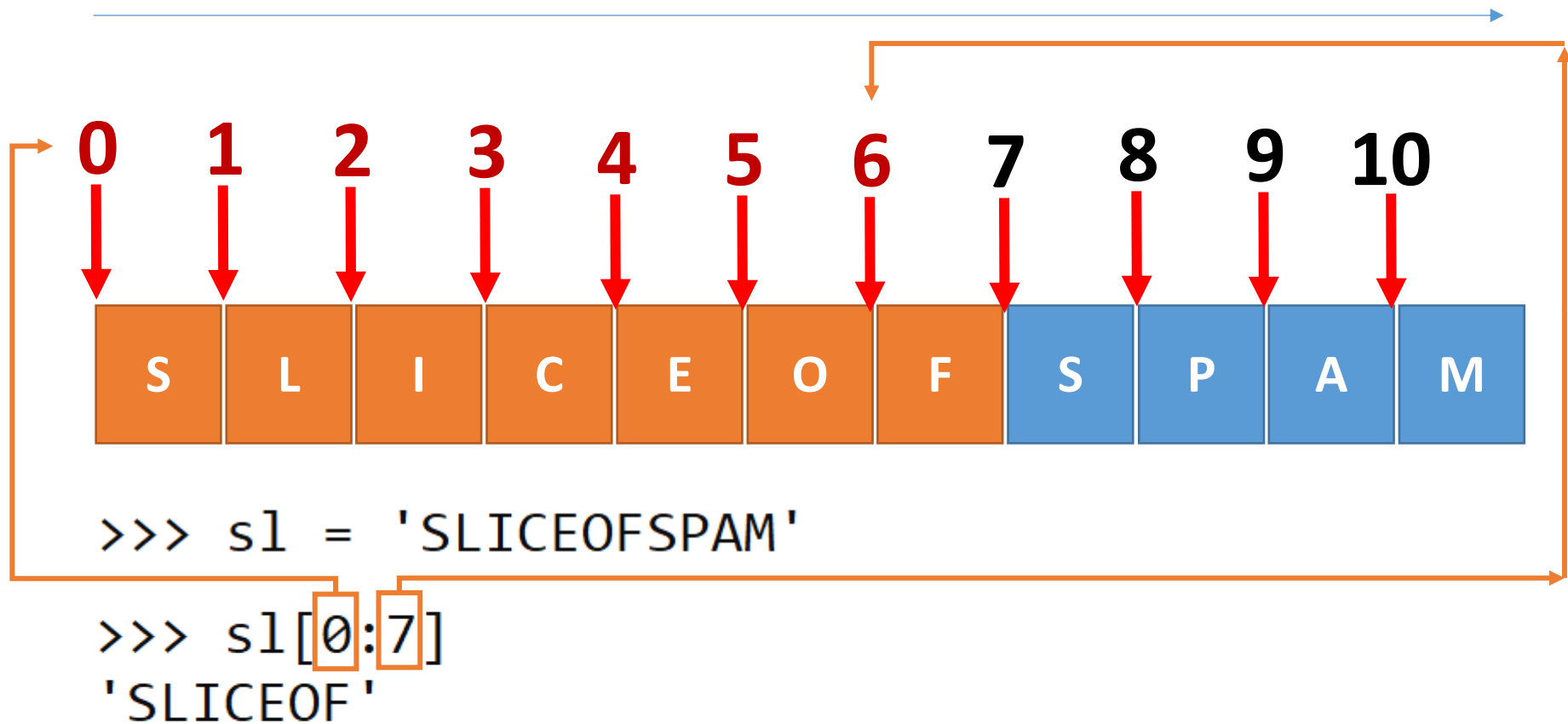
Truy cập chuỗi với Index dương



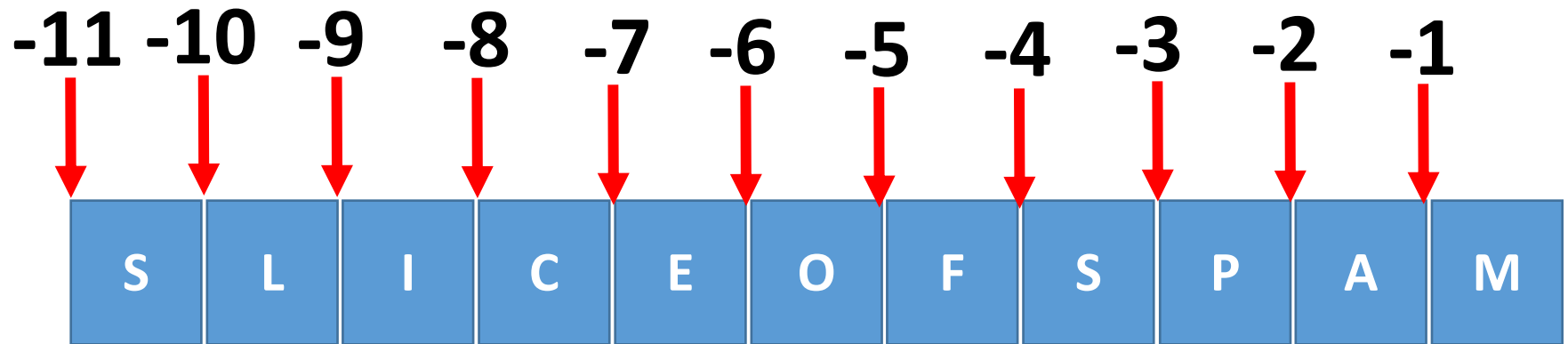
Truy cập chuỗi với Index dương



Truy cập chuỗi với Index dương

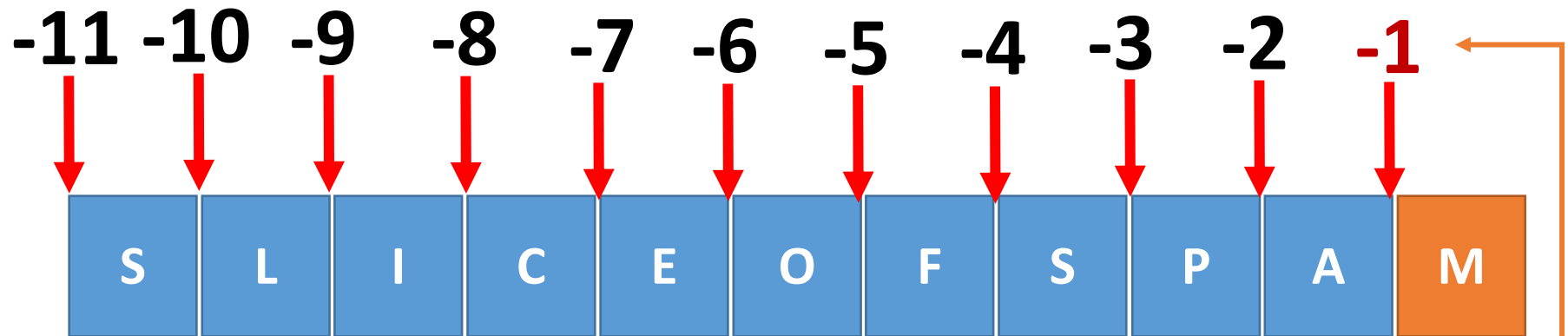


Truy cập chuỗi với Index âm



```
>>> s1 = 'SLICEOFSPAM'
```

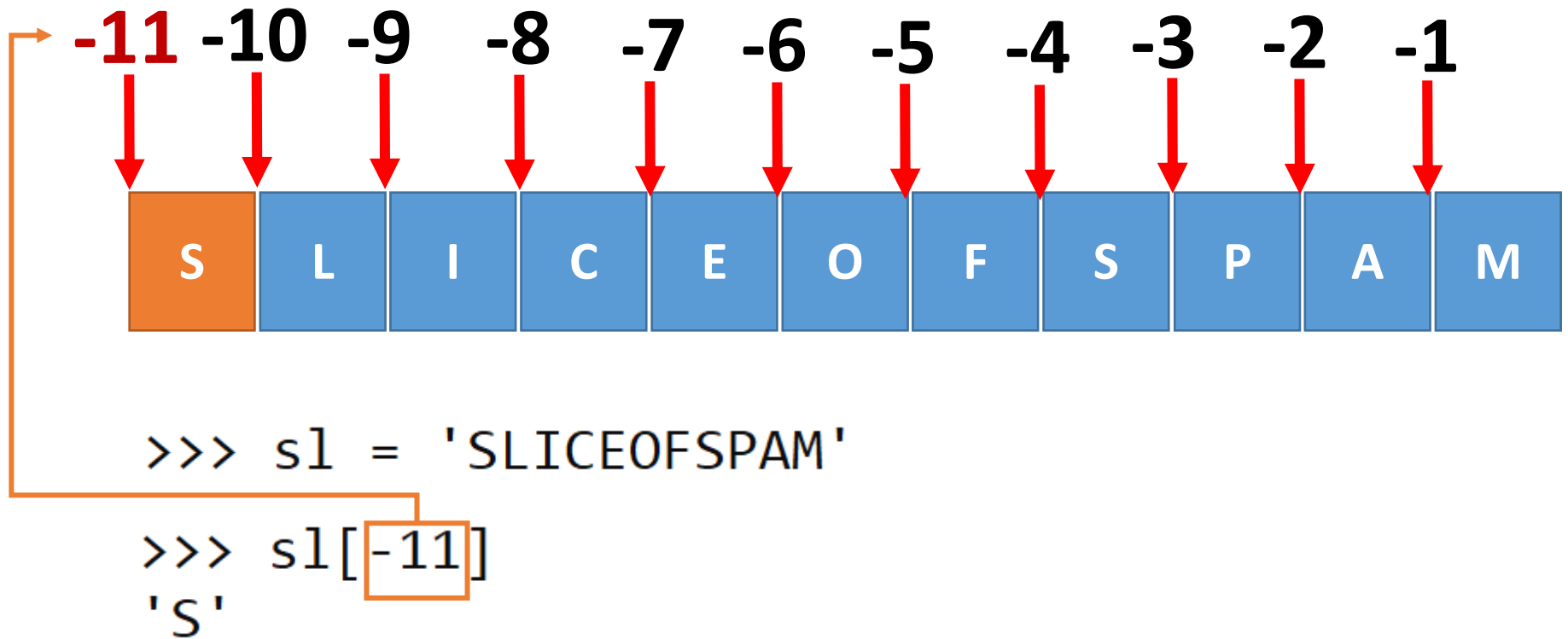
Truy cập chuỗi với Index âm



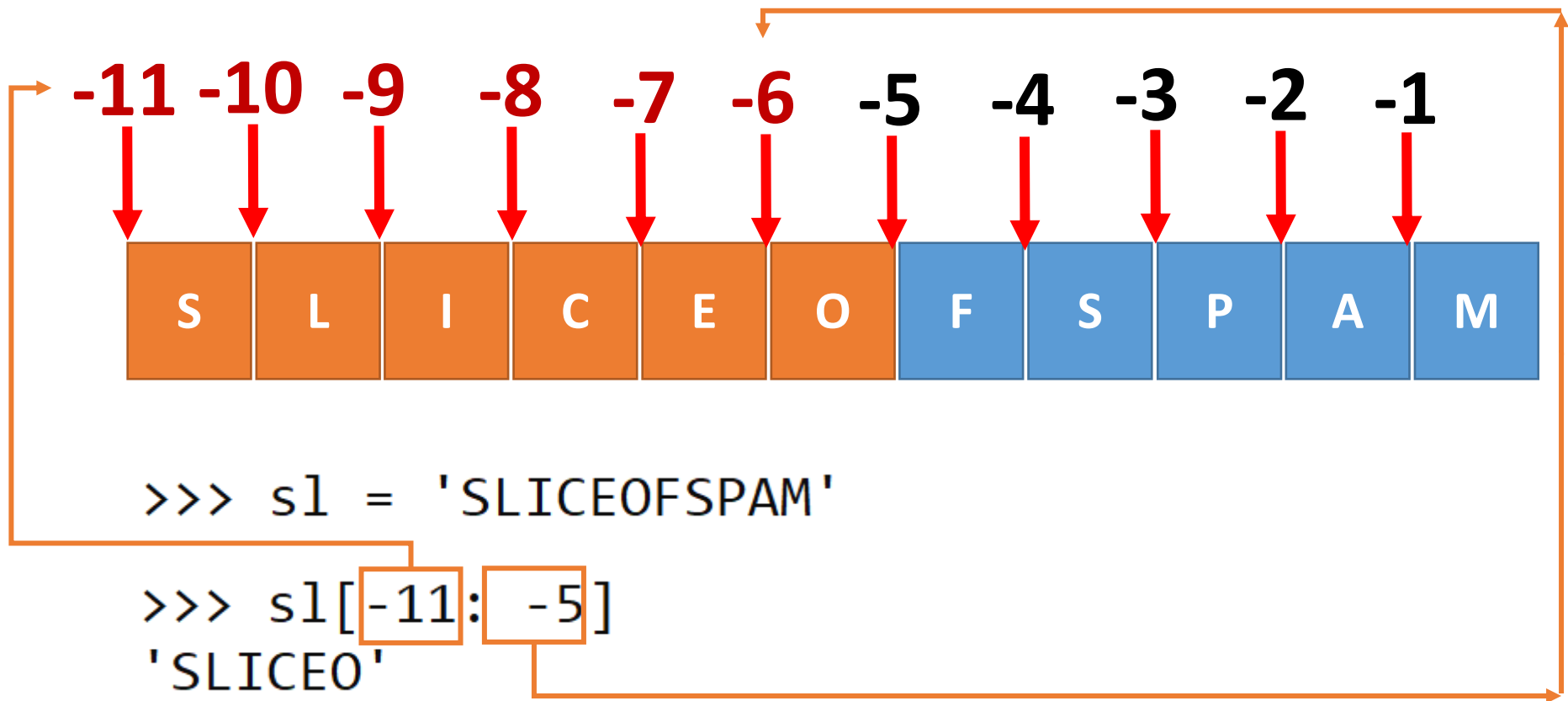
```
>>> s1 = 'SLICEOFSPAM'
```

```
>>> s1[-1]  
'M'
```

Truy cập chuỗi với Index âm



Truy cập chuỗi với Index âm



1.9.2.4 Định dạng chuỗi

- Định dạng chuỗi sử dụng **toán tử %**

- **Cú pháp:**

<chuỗi> % (<giá trị 1>, <giá trị 2>, ... <giá trị n>)

Trong đó:

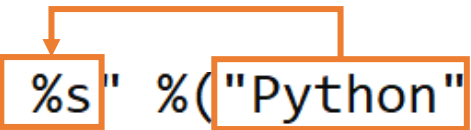
<chuỗi>: Là một giá trị chuỗi cần được định dạng

<giá trị 1> đến <giá trị n>: Là các giá trị cụ thể hoặc các biến chứa giá trị cần chèn vào chuỗi.

1.9.2.4 Định dạng chuỗi (tt)

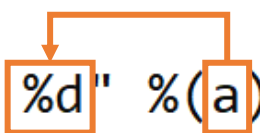
- Ví dụ:

```
>>> "Tên tôi là: %s" %("Python")  
'Tên tôi là: Python'
```



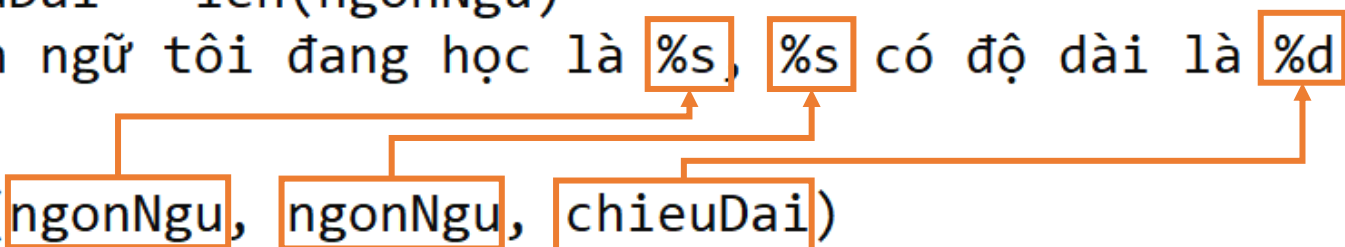
```
graph TD
    A["%s"] --> B["Python"]
```

```
>>> a = 5  
>>> "Giá trị a = %d" %(a)  
'Giá trị a = 5'
```



```
graph TD
    A["%d"] --> B["a"]
```

```
>>> ngonNgu = "Python"  
>>> chieuDai = len(ngonNgu)  
>>> "Ngôn ngữ tôi đang học là %s, %s có độ dài là %d  
ký tự" %(ngonNgu, ngonNgu, chieuDai)  
'Ngôn ngữ tôi đang học là Python, Python có độ dài là  
6 ký tự'
```



```
graph TD
    A["%s, %s có độ dài là %d ký tự"] --> B["ngonNgu"]
    A --> C["ngonNgu"]
    A --> D["chieuDai"]
```

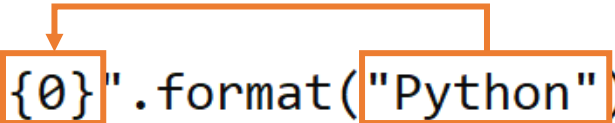
1.9.2.4 Định dạng chuỗi (tt)

- Dùng phương thức format: phương thức được python cung cấp sẵn để định dạng chuỗi.
- Cú pháp:
 - “<... {0-n} ... {0-n}>”.format(<giá trị 1>, <giá trị 2>, ... <giá trị n>)
 - Trong đó:
 - {0-n}: là một giá trị chuỗi cần định dạng.
 - <giá trị 1> đến <giá trị n>: là các giá trị cụ thể hoặc các biến sẽ chèn vào {0-n} tương ứng.

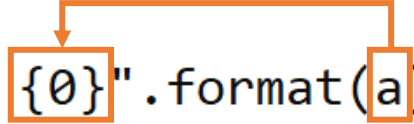
1.9.2.4 Định dạng chuỗi (tt)

- Ví dụ:

```
>>> "Tên tôi là {0}".format("Python")  
'Tên tôi là Python'
```

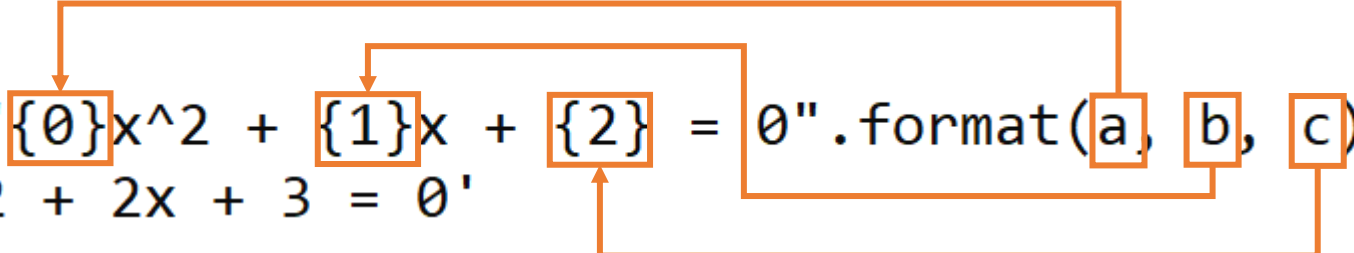


```
>>> a = 5  
>>> "Giá trị a = {0}".format(a)  
'Giá trị a = 5'
```



```
>>> a = 1; b = 2; c = 3
```

```
>>> "{0}x^2 + {1}x + {2} = 0".format(a, b, c)  
'1x^2 + 2x + 3 = 0'
```

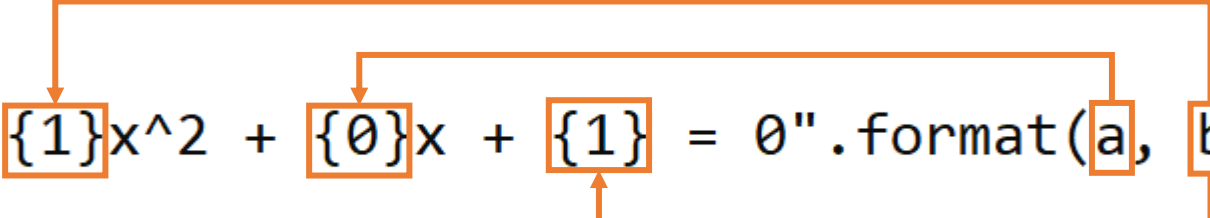


1.9.2.4 Định dạng chuỗi (tt)

- Ví dụ:

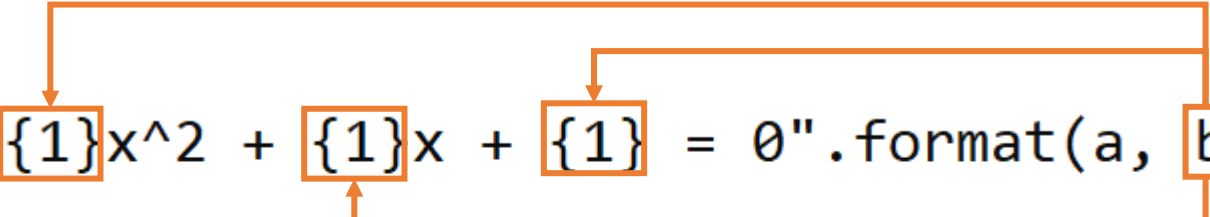
```
>>> a = 2; b = 3
```

```
>>> "{1}x^2 + {0}x + {1} = 0".format(a, b)
```



```
'3x^2 + 2x + 3 = 0'
```

```
>>> "{1}x^2 + {1}x + {1} = 0".format(a, b)
```



```
'3x^2 + 3x + 3 = 0'
```

1.9.2.5 Các phương thức

- Phương thức là một **hàm thành phần trong lớp** (class), trong lập trình hướng thủ tục, để sử dụng nó chúng ta cần gọi qua một **đối tượng cụ thể**.
 - Phương thức *count()*
 - Phương thức *capitalize()*
 - Phương thức *upper()*
 - Phương thức *lower()*
 - Phương thức *title()*
 - Phương thức *rstrip()*
 - Phương thức *strip()*
 - Phương thức *split()*
 - Phương thức *join()*
 - Phương thức *find()*

1.9.2.5 Các phương thức (tt)

Phương thức count(): Đếm **số lần xuất hiện** của một **chuỗi con** trong một chuỗi.

❖ **Cú pháp:** *<chuỗi>.count(<substr> [,start [,end]])*

- *<chuỗi>*: là một biến hoặc giá trị chuỗi
- *<substr>*: Là một chuỗi con hoặc ký tự cần đếm
- *[,start* : Là vị trí bắt đầu đếm trong chuỗi, mặc định là 0.
- *[, end]*: Là vị trí kết thúc đếm, mặc định là ký tự cuối cùng của chuỗi.

1.9.2.5 Các phương thức (tt)

Ví dụ: đếm số khoảng cách trong một chuỗi.

```
>>> a = "Lập_trình_python_không_khó"  
>>> a.count(" ")  
4
```

Ví dụ: đếm số ký tự **n** xuất hiện trong một chuỗi với index bắt đầu là 7.

```
          0   2   4   6   8   10  12  14  16  18  20  22  24  
>>> a = "Lập trình python không khó"  
          1   3   5   7   9  11  13  15  17  19  21  23  25  
  
>>> a.count("n", 7)  
3
```


1.9.2.5 Các phương thức (tt)

Ví dụ: đếm số ký tự **n** xuất hiện trong một chuỗi với index bắt đầu là 7 tới 20.

```
>>> a = "Lập trình python không khó"
      0  2  4  6  8 10 12 14 16 18 20 22 24
      1  3  5  7  9 11 13 15 17 19 21 23 25

>>> a.count("n", 7, 20)
2
```

1.9.2.5 Các phương thức (tt)

```
>>> a = "Lập trình Python không khó"
```

Phương thức *capitalize()*: viết hoa ký tự đầu tiên, các ký tự còn lại viết thường.

```
>>> a.capitalize()  
'Lập trình python không khó'
```

Phương thức *upper()*: viết hoa tất cả các ký tự của chuỗi.

```
>>> a.upper()  
'LẬP TRÌNH PYTHON KHÔNG KHÓ'
```

1.9.2.5 Các phương thức (tt)

```
>>> a = "Lập trình Python không khó"
```

Phương thức ***lower()***: *viết thường tất cả các ký tự của chuỗi.*

```
>>> a.lower()  
'lập trình python không khó'
```

Phương thức ***title()***: *viết hoa các ký tự đầu của mỗi chữ trong chuỗi.*

```
>>> a.title()  
'Lập Trình Python Không Khó'
```

1.9.2.5 Các phương thức (tt)

❖ **Phương thức `rstrip()`:** Xóa bỏ tất cả các ký tự trắng hoặc ký tự được chỉ định bên trái chuỗi.

❖ Cú pháp: `<chuỗi>.rstrip([chars])`

❖ Trong đó

- `<chuỗi>`: Là một biến hoặc giá trị chuỗi chứa ký tự cần xóa.
- `[chars]`: là ký tự cần xóa, hoặc ký tự trắng (dấu cách) nếu không chỉ định tham số này.

1.9.2.5 Các phương thức (tt)

- Ví dụ:

```
>>> s = "*****Lập trình Python*****"
>>> hoten = "    Mai Thị Lựu    "
>>> s.lstrip("*")
'Lập trình Python*****'
>>> hoten.lstrip()
'Mai Thị Lựu '

>>> s.rstrip("*")
'*****Lập trình Python'
>>> hoten.rstrip()
'Mai Thị Lựu '
```

1.9.2.5 Các phương thức (tt)

❖ **Phương thức `strip()`:** Xóa bỏ tất cả các ký tự trắng hoặc ký tự được chỉ định bên phải và bên trái của chuỗi.

❖ Cú pháp: `<chuỗi>.strip([chars])`

❖ Trong đó

- `<chuỗi>`: Là một biến hoặc giá trị chuỗi chứa ký tự cần xóa.
- `[chars]`: là ký tự cần xóa, hoặc ký tự trắng (dấu cách) nếu không chỉ định tham số này.

1.9.2.5 Các phương thức (tt)

- Ví dụ:

```
>>> s = "*****Lập trình Python*****"
>>> hoten = "    Mai Thị Lựu    "
>>> s.strip("*")
'Lập trình Python'
>>> hoten.strip()
'Mai Thị Lựu'
```

1.9.2.5 Các phương thức (tt)

❖ **Phương thức `split()`:** Tách một chuỗi thành nhiều chuỗi con, giá trị trả về là một danh sách.

❖ Cú pháp: `<chuỗi>.split([chars],[n])`

❖ Trong đó

- `<chuỗi>`: Là một biến hoặc giá trị chuỗi chứa ký tự cần xóa.
- `[chars]`: là ký tự ngăn cách các chuỗi con, mặc định là khoảng trắng.
- `n`: chỉ định số lần tách, mặc định là -1

1.9.2.5 Các phương thức (tt)

- Ví dụ:

```
>>> s = "Khoa Công nghệ thông tin"
```

```
>>> s.split()  
['Khoa', 'Công', 'nghệ', 'thông', 'tin']
```

```
>>> s.split("t")  
['Khoa Công nghệ ', 'hông ', 'in']
```

```
>>> s.split("t", 1)  
['Khoa Công nghệ ', 'hông tin']
```

```
.....
```

1.9.2.5 Các phương thức (tt)

❖ **Phương thức join():** Nối các chuỗi con thành một chuỗi.

Các chuỗi con có thể là: tuple, list, set,... Giá trị trả về là một chuỗi.

❖ Cú pháp: *<ký tự nối>.join(<tham số>)*

❖ Trong đó

- *<ký tự nối>*: Là ký tự được nối với từng chuỗi con.
- *[tham số]*: là một danh sách, tập hợp chứa các chuỗi con cần nối.

1.9.2.5 Các phương thức (tt)

- Ví dụ:

```
>>> s = ("Khoa", "Công", "Nghệ", "Thông", "Tin")
```

```
>>> khoa = "_".join(s)
```

```
>>> khoa
```

```
'Khoa_Công_Nghệ_Thông_Tin'
```

```
>>> khoa = "-".join(s)
```

```
>>> khoa
```

```
'Khoa-Công-Nghệ-Thông-Tin'
```

```
>>> khoa = "".join(s)
```

```
>>> khoa
```

```
'KhoaCôngNghệThôngTin'
```

1.9.2.5 Các phương thức (tt)

❖ **Phương thức find():** Tìm kiếm chuỗi con trong 1 chuỗi. Trả về vị trí đầu tiên chuỗi con xuất hiện trong chuỗi, nếu không tìm thấy giá trị trả về là -1.

❖ Cú pháp: *<chuỗi>.find(substr [,start [,end]])*

❖ Trong đó

- *<chuỗi>*: Là một biến hoặc giá trị chuỗi chứa chuỗi cần tìm kiếm.
- *substr*: Là một chuỗi con hoặc giá trị chuỗi cần tìm kiếm trong *<chuỗi>*
- *[,start]*: vị trí bắt đầu tìm kiếm, mặc định là 0
- *[, end]*: vị trí kết thúc tìm kiếm, mặc định là tìm kiếm đến cuối chuỗi.

1.9.2.5 Các phương thức (tt)

- Ví dụ:

```
>>> s = "Lập trình không khó"  
>>> s.find("trình")  
4  
>>> s.find("không", 2, 10)  
-1
```

1.9.3 Lists

- Lists are **positionally ordered collections** of arbitrarily typed objects, and they have no fixed size.
- They are also **mutable** - unlike strings, lists can be **modified** in-place by assignment to offsets as well as a variety of list method calls.

1.9.3.1 Sequence Operations

- lists **support all the sequence operations** we discussed for strings.
- **difference** is that the results are usually **lists** instead of strings

1.9.3.1 Sequence Operations

```
Command Prompt - python

>>>
>>> L = [123, 'Python', 1.23]
>>> len(L)
3
>>> L[0]
123
>>> L[:-1]
[123, 'Python']
>>> L + [4, 5, 6]
[123, 'Python', 1.23, 4, 5, 6]
>>> L
[123, 'Python', 1.23]
>>>
```

**# A list of three
different-type
objects**

1.9.3.1 Sequence Operations (con't)

```
Command Prompt - python

>>>
>>> L = [123, 'Python', 1.23]
>>> len(L)
3
>>> L[0]
123
>>> L[:-1]
[123, 'Python']
>>> L + [4, 5, 6]
[123, 'Python', 1.23, 4, 5, 6]
>>> L
[123, 'Python', 1.23]
>>>
```

**Number of items
in the list**

1.9.3.1 Sequence Operations (con't)

```
Command Prompt - python

>>>
>>> L = [123, 'Python', 1.23]
>>> len(L)
3
>>> L[0]
123
>>> L[:-1]
[123, 'Python']
>>> L + [4, 5, 6]
[123, 'Python', 1.23, 4, 5, 6]
>>> L
[123, 'Python', 1.23]
>>>
```

**Indexing by
Position**

1.9.3.1 Sequence Operations (con't)

```
Command Prompt - python

>>>
>>> L = [123, 'Python', 1.23]
>>> len(L)
3
>>> L[0]
123
>>> L[:-1]
[123, 'Python']
>>> L + [4, 5, 6]
[123, 'Python', 1.23, 4, 5, 6]
>>> L
[123, 'Python', 1.23]
>>>
```

**Slicing a list
returns a new list**

1.9.3.1 Sequence Operations (con't)

```
Command Prompt - python

>>>
>>> L = [123, 'Python', 1.23]
>>> len(L)
3
>>> L[0]
123
>>> L[:-1]
[123, 'Python']
>>> L + [4, 5, 6]
[123, 'Python', 1.23, 4, 5, 6]
>>> L
[123, 'Python', 1.23]
>>>
```

**Concatenation
makes a new list
too**

1.9.3.1 Sequence Operations (con't)

```
Command Prompt - python

>>>
>>> L = [123, 'Python', 1.23]
>>> len(L)
3
>>> L[0]
123
>>> L[:-1]
[123, 'Python']
>>> L + [4, 5, 6]
[123, 'Python', 1.23, 4, 5, 6]
>>> L
[123, 'Python', 1.23]
>>>
```

**We're not
changing the
original list**

1.9.3.1 Type-Specific Operations

- Lists have no **fixed size**.
- List can **grow** and **shrink** on demand

Growing: add object
at end of list

```
Command Prompt - python

>>>
>>> L.append("HCM")
>>> L
[123, 'Python', 1.23, 'HCM']
>>> L.pop(2)
1.23
>>> L
[123, 'Python', 'HCM']
>>> L.sort()
```

1.9.3.1 Type-Specific Operations (con't)

- Lists have no **fixed size**.
- List can **grow** and **shrink** on demand

Shrinking: delete an item in the middle

```
Command Prompt - python

>>>
>>> L.append("HCM")
>>> L
[123, 'Python', 1.23, 'HCM']
>>> L.pop(2)
1.23
>>> L
[123, 'Python', 'HCM']
```

1.9.3.1 Type-Specific Operations (con't)

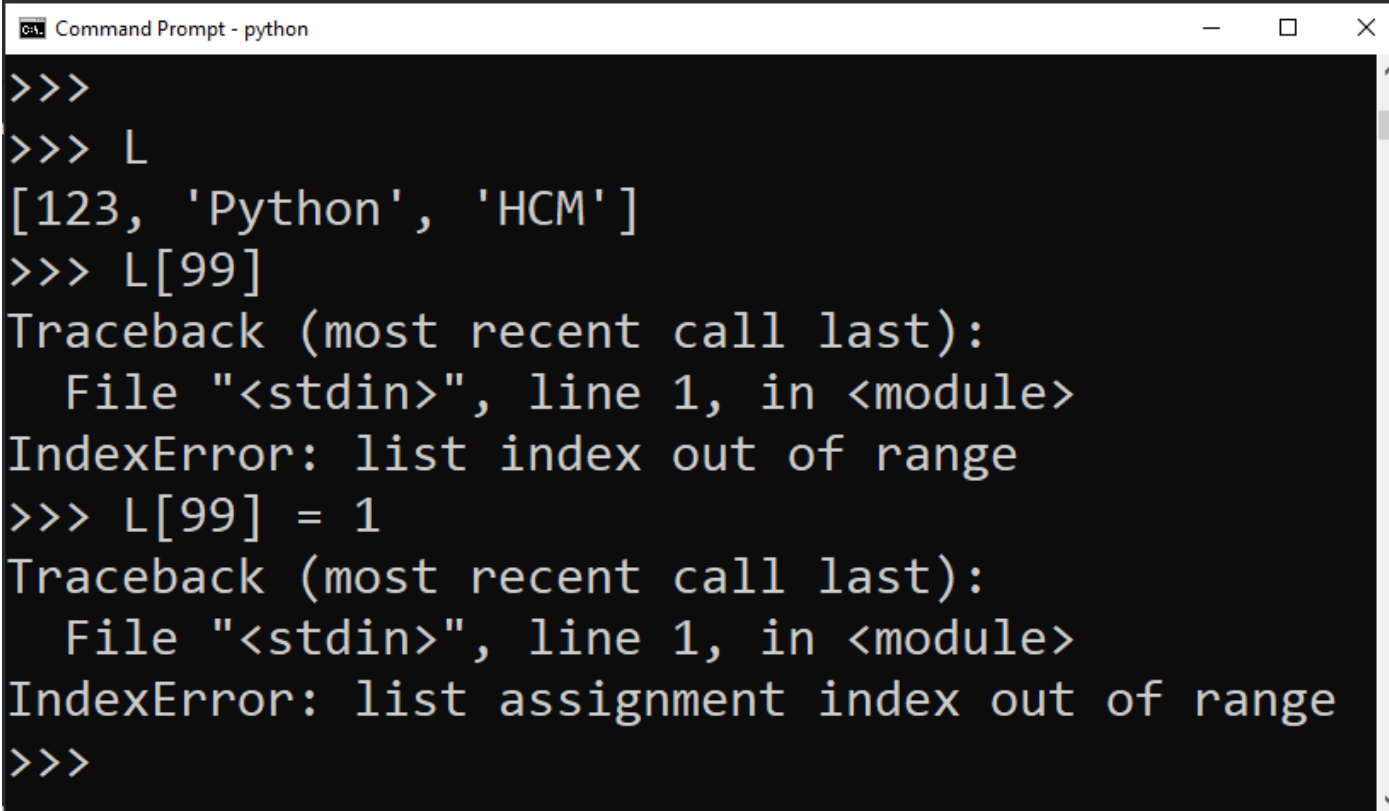
- Lists have no **fixed size**.
- List can **grow** and **shrink** on demand

"del L[2]" deletes
from a list too

```
Command Prompt - python
>>>
>>> L.append("HCM")
>>> L
[123, 'Python', 1.23, 'HCM']
>>> L.pop(2)
1.23
>>> L
[123, 'Python', 'HCM']
```


1.9.3.2 Bounds Checking

- Python still **doesn't allow** us to reference items that are **not present**

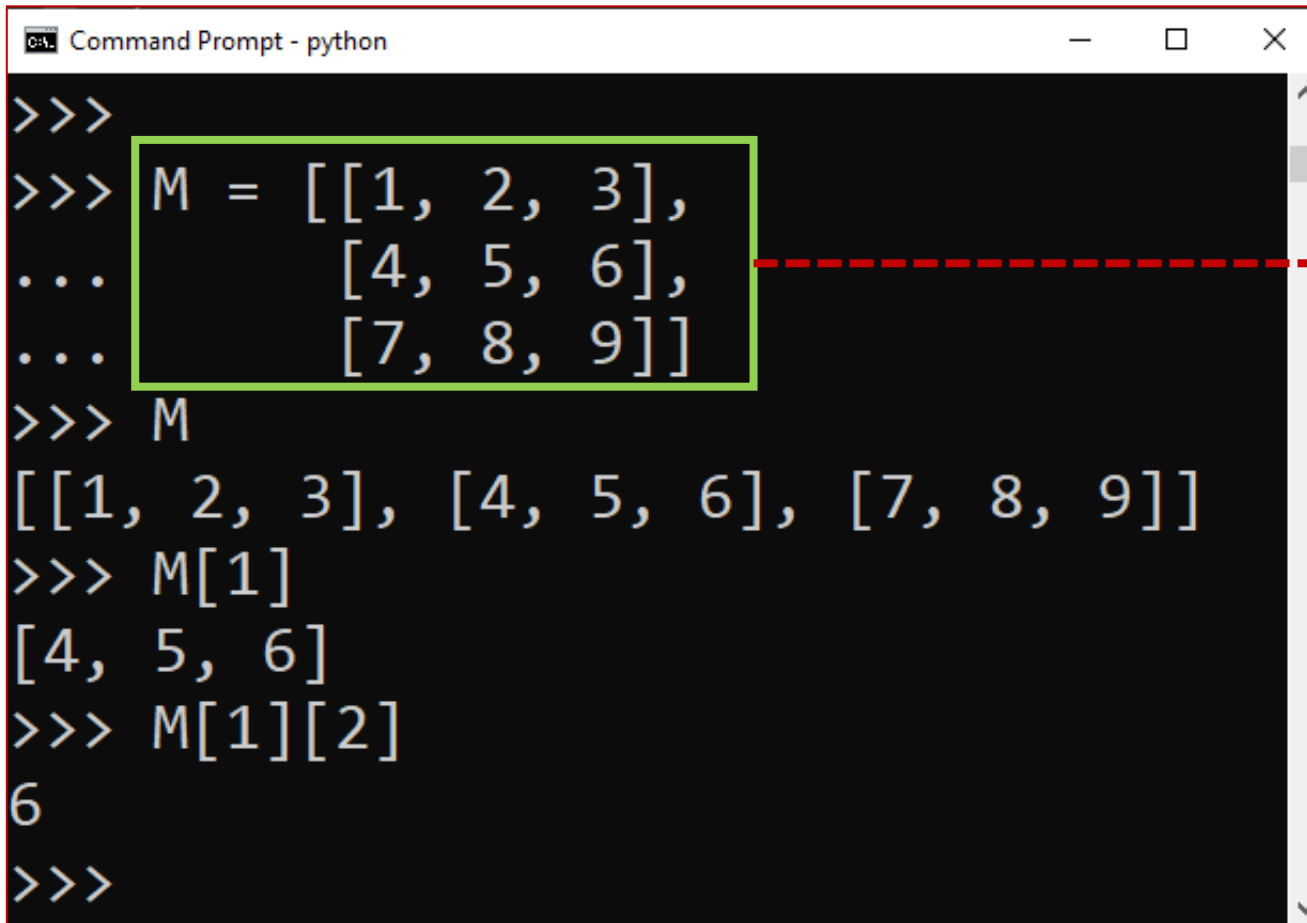


```
Command Prompt - python
>>>
>>> L
[123, 'Python', 'HCM']
>>> L[99]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> L[99] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>>
```

1.9.3.4 Nesting

support **arbitrary nesting**, we can nest them in any combination

A 3×3 matrix, as
nested lists
Code can span lines if
bracketed



```
Command Prompt - python
>>>
>>> M = [[1, 2, 3],
...      [4, 5, 6],
...      [7, 8, 9]]
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> M[1]
[4, 5, 6]
>>> M[1][2]
6
>>>
```

The screenshot shows a Python Command Prompt window. The first three lines of code define a 3x3 matrix M as a list of lists: M = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]. The fourth line prints M, showing the full matrix. The fifth line prints M[1], showing the second row [4, 5, 6]. The sixth line prints M[1][2], showing the element 6. A green box highlights the matrix definition code, and a red dashed line connects it to the text box on the right.

1.9.3.4 Nesting

support **arbitrary nesting**, we can nest them in any combination

Get Row #2

```
Command Prompt - python
>>>
>>> M = [[1, 2, 3],
...      [4, 5, 6],
...      [7, 8, 9]]
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> M[1]
[4, 5, 6]
>>> M[1][2]
6
>>>
```

1.9.3.4 Nesting

support **arbitrary nesting**, we can nest them in any combination

Get row 2, then
get item 3 within
the row

```
Command Prompt - python
>>>
>>> M = [[1, 2, 3],
...       [4, 5, 6],
...       [7, 8, 9]]
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> M[1]
[4, 5, 6]
>>> M[1][2]
6
>>>
```

1.9.3.4 Comprehensions

- Python includes a more advanced operation known as a list comprehension expression, which turns out to be a powerful way to process structures like our matrix.

```
Select Command Prompt - python

>>>
>>> col2 = [row[1] for row in M]
>>> col2
[2, 5, 8]
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>>
```

**Collect the items
in column 2**

1.9.3.5 Comprehensions

- Python includes a more advanced operation known as a list comprehension expression, which turns out to be a powerful way to process structures like our matrix.

```
Select Command Prompt - python
>>>
>>> col2 = [row[1] for row in M]
>>> col2
[2, 5, 8]
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>>
```

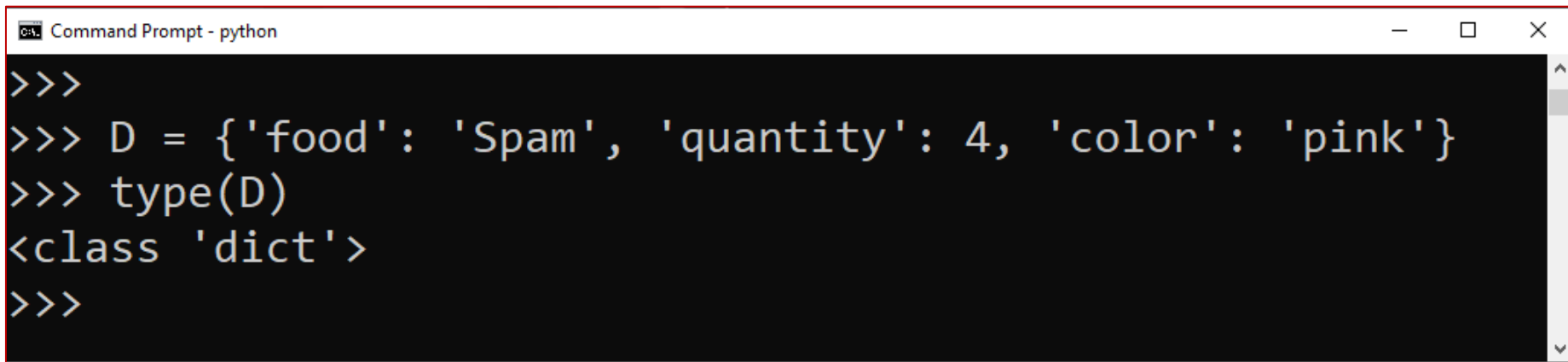
**The matrix is
unchanged**

1.9.4 Dictionaries

- Dictionaries are **not sequences** at all, but are instead known as **mappings**.
- Mappings store objects by key instead of by relative position
- Simply map **keys** to associated **values**.
- Dictionaries are mutable (be changed in-place; grow and shrink on demand).

1.9.4 Mapping Operations

- dictionaries are coded in **curly braces** and consist of a series of **“key: value” pairs**.
- useful anytime we need to **associate** a **set of values with keys**

A screenshot of a Windows Command Prompt window titled "Command Prompt - python". The window has a black background with white text. The text shows a Python interactive session where a dictionary 'D' is created with three key-value pairs: 'food': 'Spam', 'quantity': 4, and 'color': 'pink'. The type of 'D' is then checked, returning '<class 'dict'>'.

```
>>>  
>>> D = {'food': 'Spam', 'quantity': 4, 'color': 'pink'}  
>>> type(D)  
<class 'dict'>  
>>>
```


1.9.4 Mapping Operations (con't)

- **index** this dictionary **by key** to fetch and change the keys' associated values.

Fetch value of
key 'food'

```
Command Prompt - python
>>>
>>> D['food']
'Spam'
>>> D['quantity'] += 1
>>> D
{'food': 'Spam', 'quantity': 5, 'color': 'pink'}
>>>
```

1.9.4 Mapping Operations (con't)

- **index** this dictionary **by key** to fetch and change the keys' associated values.

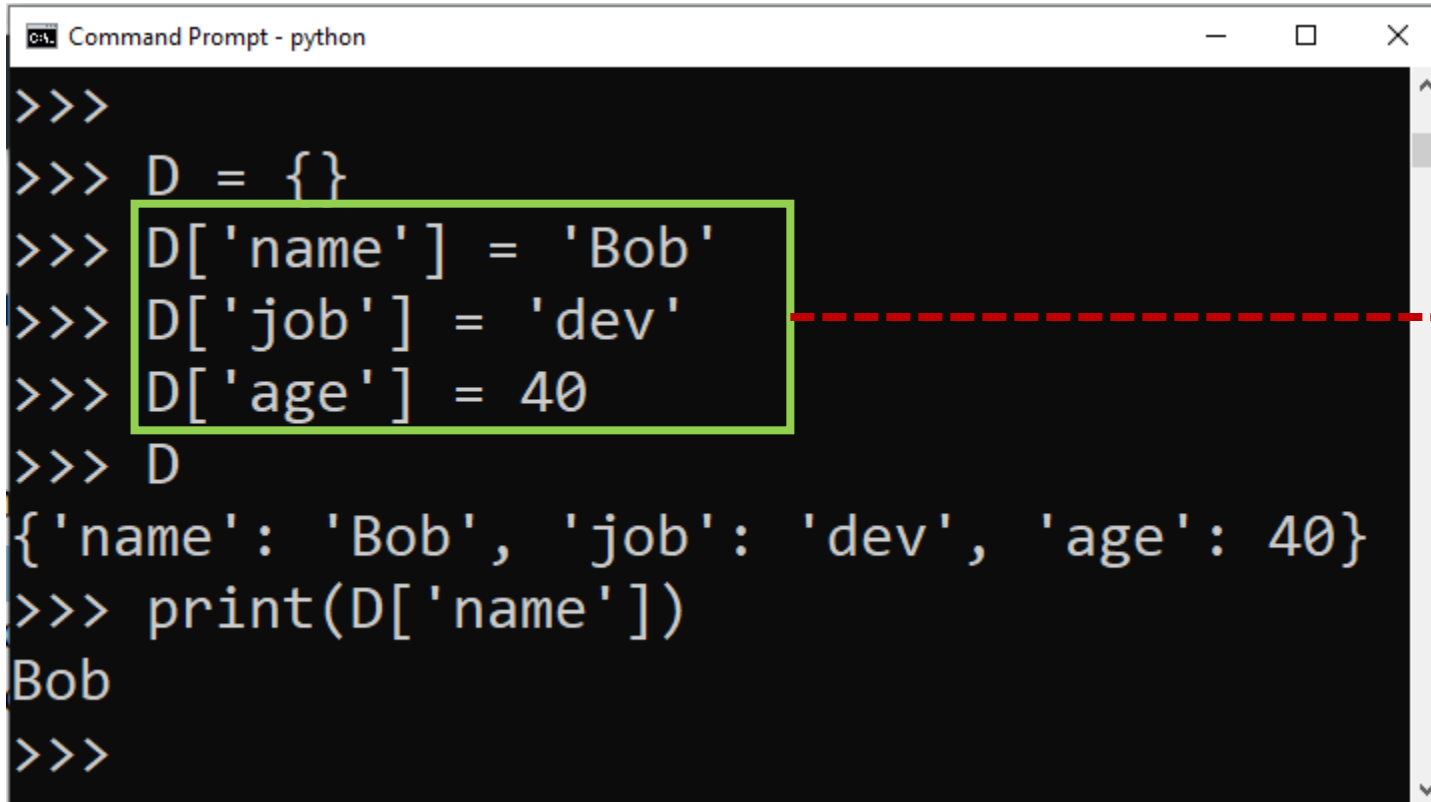
Add 1 to
'quantity' value

```
Command Prompt - python
>>>
>>> D['food']
'Spam'
>>> D['quantity'] += 1
>>> D
{'food': 'Spam', 'quantity': 5, 'color': 'pink'}
```

1.9.4 Mapping Operations (con't)

curly-braces literal form does see use, it is perhaps more common to see dictionaries built up in different ways.

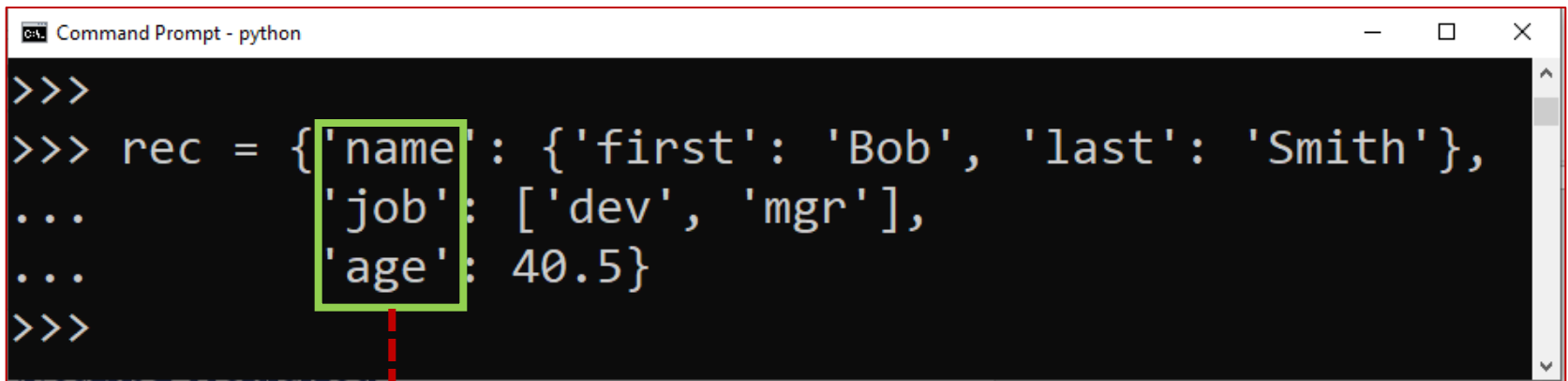
**Create keys by
assignment**



```
Command Prompt - python
>>>
>>> D = {}
>>> D['name'] = 'Bob'
>>> D['job'] = 'dev'
>>> D['age'] = 40
>>> D
{'name': 'Bob', 'job': 'dev', 'age': 40}
>>> print(D['name'])
Bob
>>>
```

1.9.4 Nesting Revisited

- Used a dictionary to describe a hypothetical person, with three keys.

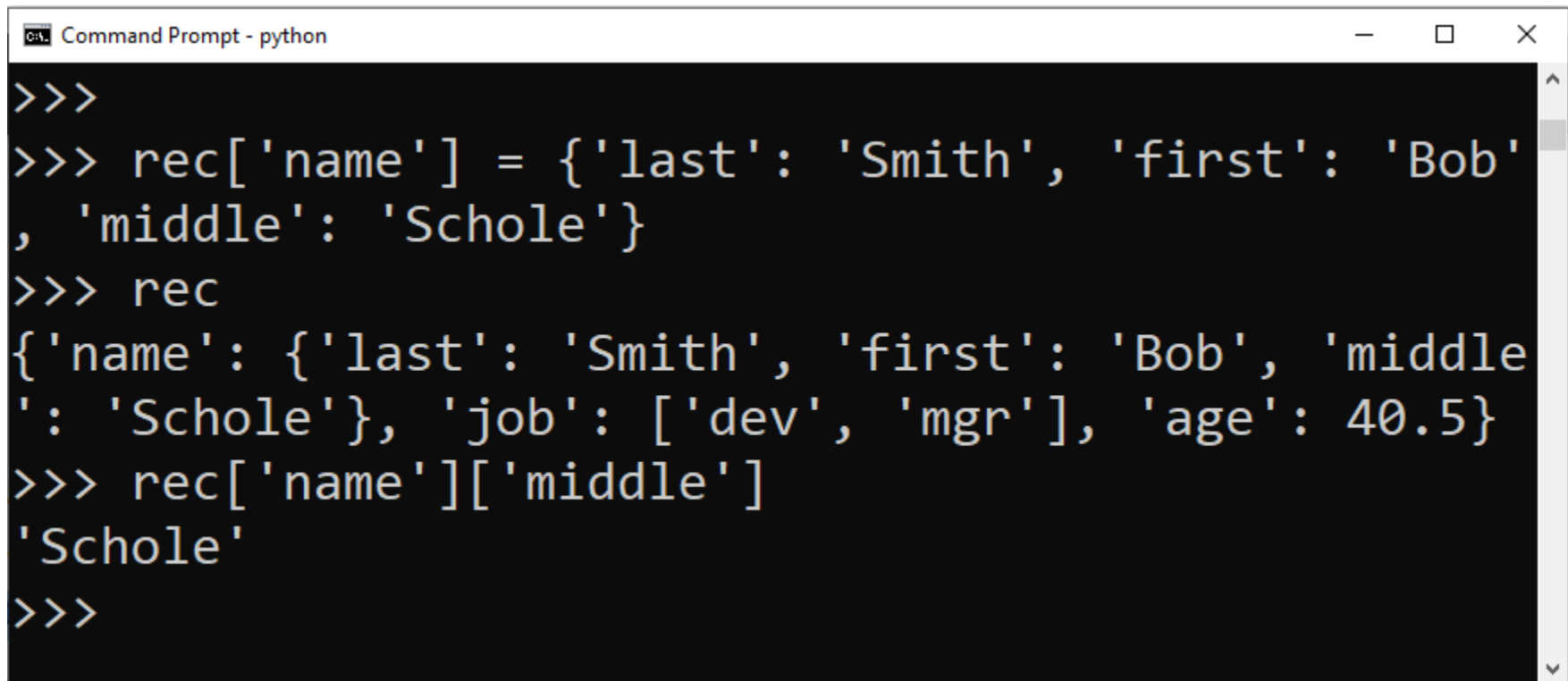


```
>>>
>>> rec = {'name': {'first': 'Bob', 'last': 'Smith'},
...       'job': ['dev', 'mgr'],
...       'age': 40.5}
>>>
```

Three top key

1.9.4 Nesting Revisited (con't)

- become more complex: a nested dictionary for the name to support multiple parts



```
>>>
>>> rec['name'] = {'last': 'Smith', 'first': 'Bob',
, 'middle': 'Schole'}
>>> rec
{'name': {'last': 'Smith', 'first': 'Bob', 'middle': 'Schole'}, 'job': ['dev', 'mgr'], 'age': 40.5}
>>> rec['name']['middle']
'Schole'
>>>
```

1.9.4 Sorting Keys: for Loops

- dictionaries are not sequences, they don't maintain any dependable left-to-right order.

```
C:\Python27\python.exe

>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D
{'a': 1, 'c': 3, 'b': 2}
>>>
```

```
Python 3.7 (64-bit)

>>>
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D
{'a': 1, 'b': 2, 'c': 3}
>>>
```

#

1.9.4 Sorting Keys: for Loops (con't)

```
C:\Python27\python.exe
>>>
>>> Ks = list(D.keys())
>>> Ks
['a', 'c', 'b']
>>> Ks.sort()
>>> Ks
['a', 'b', 'c']
>>> for key in Ks:
...     print(key, '=>', D[key])
...
('a', '=>', 1)
('b', '=>', 2)
('c', '=>', 3)
>>>
```

**Unordered Key
lists**

1.9.4 Sorting Keys: for Loops (con't)

```
C:\Python27\python.exe
>>>
>>> Ks = list(D.keys())
>>> Ks
['a', 'c', 'b']
>>> Ks.sort()
>>> Ks
['a', 'b', 'c']
>>> for key in Ks:
...     print(key, '=>', D[key])
...
('a', '=>', 1)
('b', '=>', 2)
('c', '=>', 3)
>>>
```

**Sorted Keys
lists**

1.9.4 Sorting Keys: for Loops (con't)

```
C:\Python27\python.exe
>>>
>>> Ks = list(D.keys())
>>> Ks
['a', 'c', 'b']
>>> Ks.sort()
>>> Ks
['a', 'b', 'c']
>>> for key in Ks:
...     print(key, '=>', D[key])
...
('a', '=>', 1)
('b', '=>', 2)
('c', '=>', 3)
>>>
```

Iterate though
sorted keys

1.9.4 Missing Keys: if Tests

- **fetching** a **nonexistent key** is still a mistake

Assigning new keys
grows dictionaries

```
Python 3.7 (64-bit)
>>>
>>> D
{'a': 1, 'b': 2, 'c': 3}
>>> D['e'] = 99
>>> D
{'a': 1, 'b': 2, 'c': 3, 'e': 99}
>>> D['f']
File "<stdin>", line 1
    D['f']
    ^
SyntaxError: EOL while scanning string literal
>>>
```

1.9.4 Missing Keys: if Tests (con't)

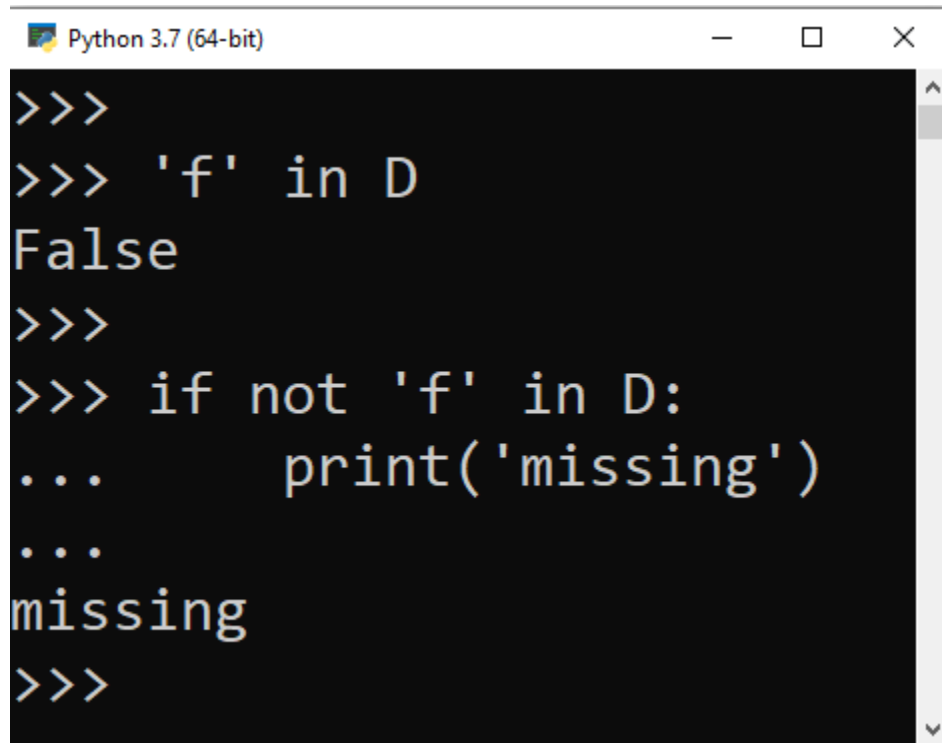
- **fetching** a **nonexistent key** is still a mistake

Referencing a nonexistent key is an error

```
Python 3.7 (64-bit)
>>>
>>> D
{'a': 1, 'b': 2, 'c': 3}
>>> D['e'] = 99
>>> D
{'a': 1, 'b': 2, 'c': 3, 'e': 99}
>>> D['f']
File "<stdin>", line 1
    D['f']
      ^
SyntaxError: EOL while scanning string literal
>>>
```

1.9.4 Missing Keys: if Tests (con't)

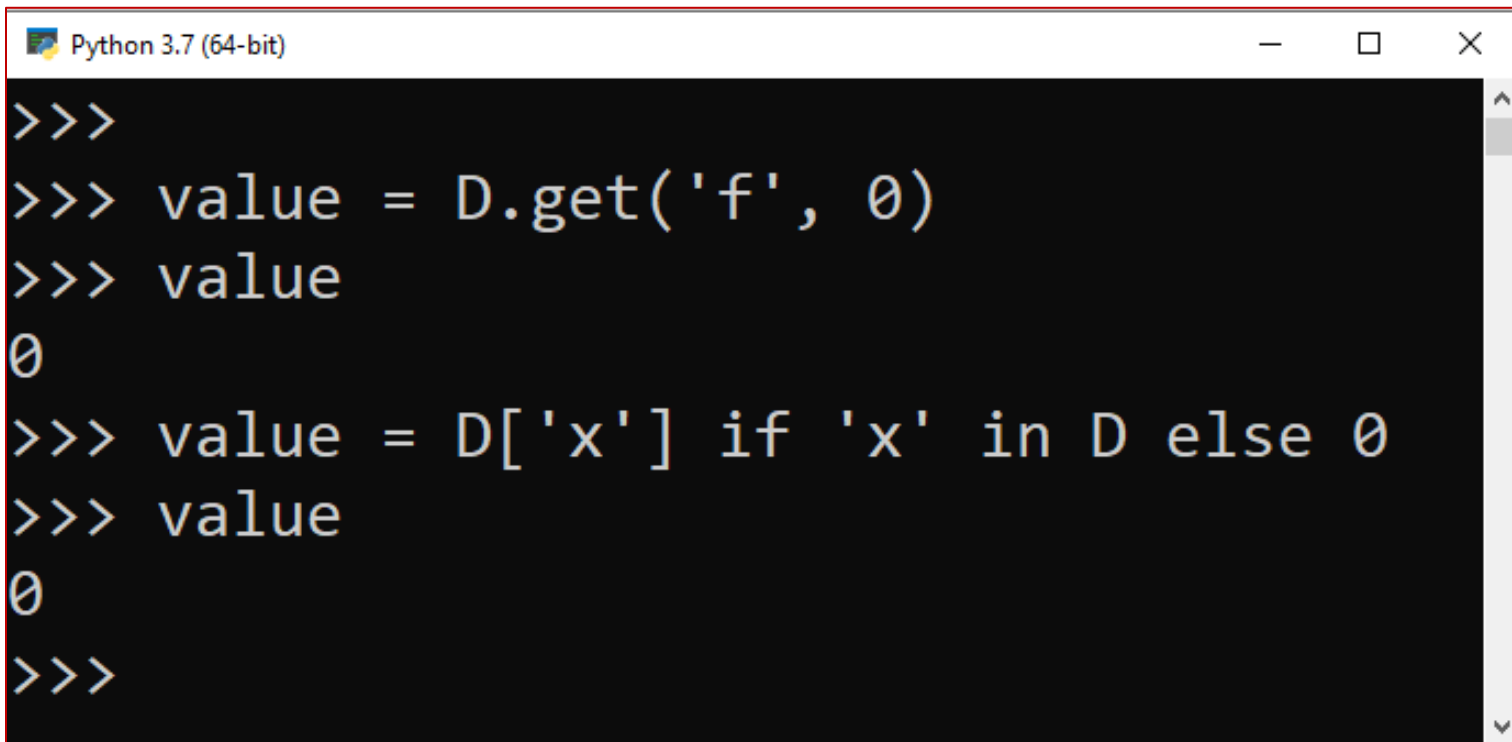
The dictionary in membership expression allows us **to query the existence of a key and branch** on the result with a Python if statement

A screenshot of a Python 3.7 (64-bit) terminal window. The window has a title bar with the text "Python 3.7 (64-bit)" and standard window controls (minimize, maximize, close). The terminal background is black with white text. The code entered is as follows:

```
>>>  
>>> 'f' in D  
False  
>>>  
>>> if not 'f' in D:  
...     print('missing')  
...  
missing  
>>>
```

1.9.4 Missing Keys: if Tests (con't)

- avoid accessing nonexistent keys: the *get* method
- the if/else expression

A screenshot of a Python 3.7 (64-bit) shell window. The window has a title bar with the Python logo and text "Python 3.7 (64-bit)", and standard window controls (minimize, maximize, close). The shell has a black background with white text. It shows a series of commands and their outputs: three prompt characters ">>>" followed by "value = D.get('f', 0)", then "value" followed by the output "0". Then another set of "value = D['x'] if 'x' in D else 0", followed by "value" and the output "0". The prompt ">>>" appears again at the end.

```
>>>
>>> value = D.get('f', 0)
>>> value
0
>>> value = D['x'] if 'x' in D else 0
>>> value
0
>>>
```

1.9.5 Tuples

- The tuple object like a list that **cannot be changed**
- Tuples are **sequences**
- Tuple are **immutable**
- Coded in **parentheses ()** instead of square bracket { }
- Support arbitrary types, arbitrary nesting, and the usual sequence operations.

1.9.5 Tuples (con't)

A 4-item tuple

```
Python 3.7 (64-bit)
>>>
>>> T = (1, 2, 3, 4)
>>> len(T)
4
>>> type(T)
<class 'tuple'>
>>> T + (5, 6)
(1, 2, 3, 4, 5, 6)
>>> T[0]
1
>>>
```

1.9.5 Tuples (con't)

Length of tuple T

```
Python 3.7 (64-bit)
>>>
>>> T = (1, 2, 3, 4)
>>> len(T)
4
>>> type(T)
<class 'tuple'>
>>> T + (5, 6)
(1, 2, 3, 4, 5, 6)
>>> T[0]
1
>>>
```


1.9.5 Tuples (con't)

```
Python 3.7 (64-bit)
>>>
>>> T = (1, 2, 3, 4)
>>> len(T)
4
>>> type(T)
<class 'tuple'>
>>> T + (5, 6)
(1, 2, 3, 4, 5, 6)
>>> T[0]
1
>>>
```

Type of T

1.9.5 Tuples (con't)

```
Python 3.7 (64-bit)
>>>
>>> T = (1, 2, 3, 4)
>>> len(T)
4
>>> type(T)
<class 'tuple'>
>>> T + (5, 6)
(1, 2, 3, 4, 5, 6)
>>> T[0]
1
>>>
```

Concatenation



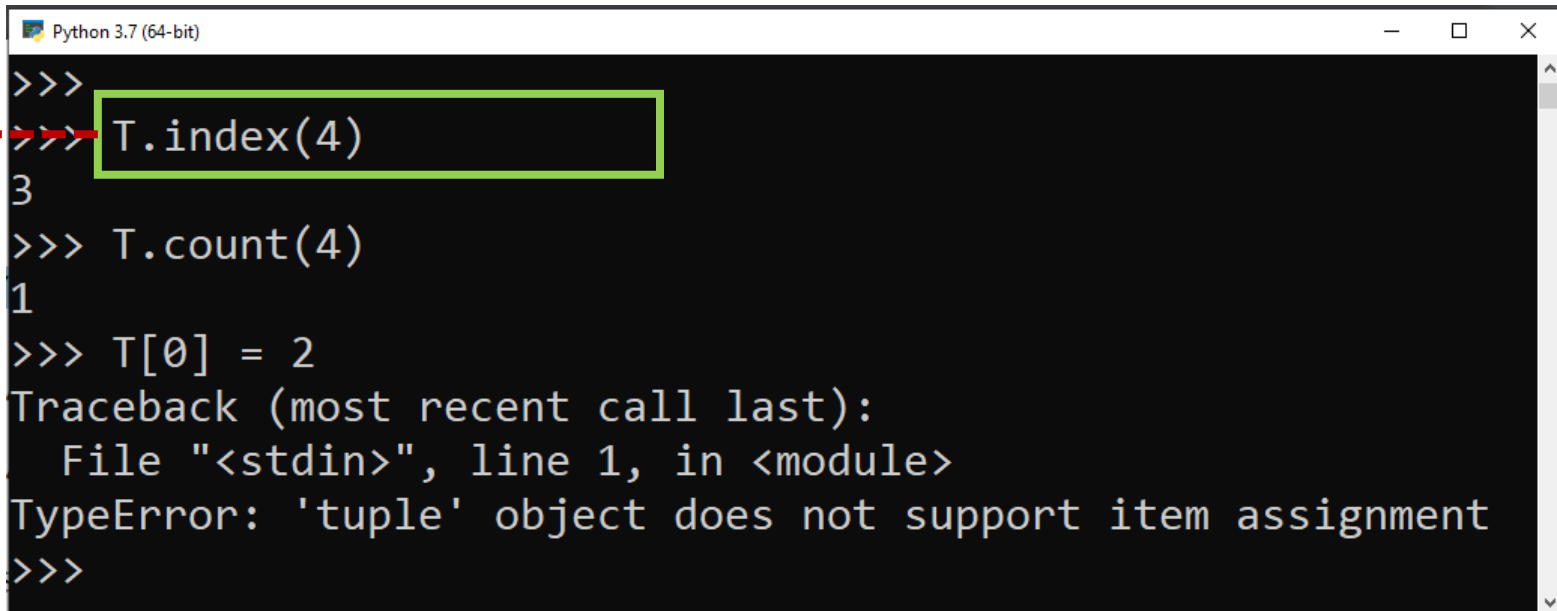
1.9.5 Tuples (con't)

```
Python 3.7 (64-bit)
>>>
>>> T = (1, 2, 3, 4)
>>> len(T)
4
>>> type(T)
<class 'tuple'>
>>> T + (5, 6)
(1, 2, 3, 4, 5, 6)
>>> T[0]
1
>>>
```

Indexing, slicing,
and more

1.9.5 Tuples (con't)

- Tuples also have two type-specific callable methods



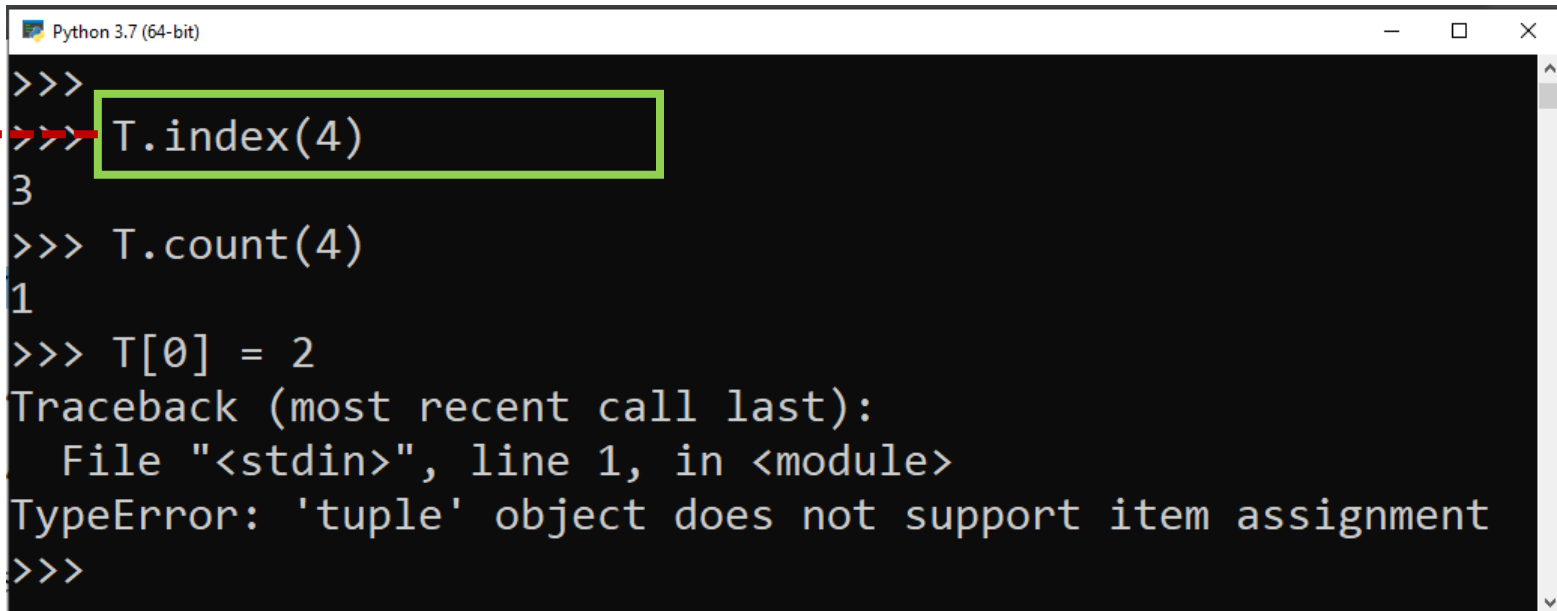
```
Python 3.7 (64-bit)
>>>
>>> T.index(4)
3
>>> T.count(4)
1
>>> T[0] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

A screenshot of a Python 3.7 (64-bit) terminal window. The window has a title bar with standard OS controls. The terminal shows a series of commands and their outputs. The first command is `T.index(4)`, which returns `3`. The second command is `T.count(4)`, which returns `1`. The third command is `T[0] = 2`, which results in a `TypeError: 'tuple' object does not support item assignment`. A red dashed line originates from the `T.index(4)` line and points to a text box on the right.

**Tuple methods: 4
appears at offset 3**

1.9.5 Tuples (con't)

- Tuples also have two type-specific callable methods



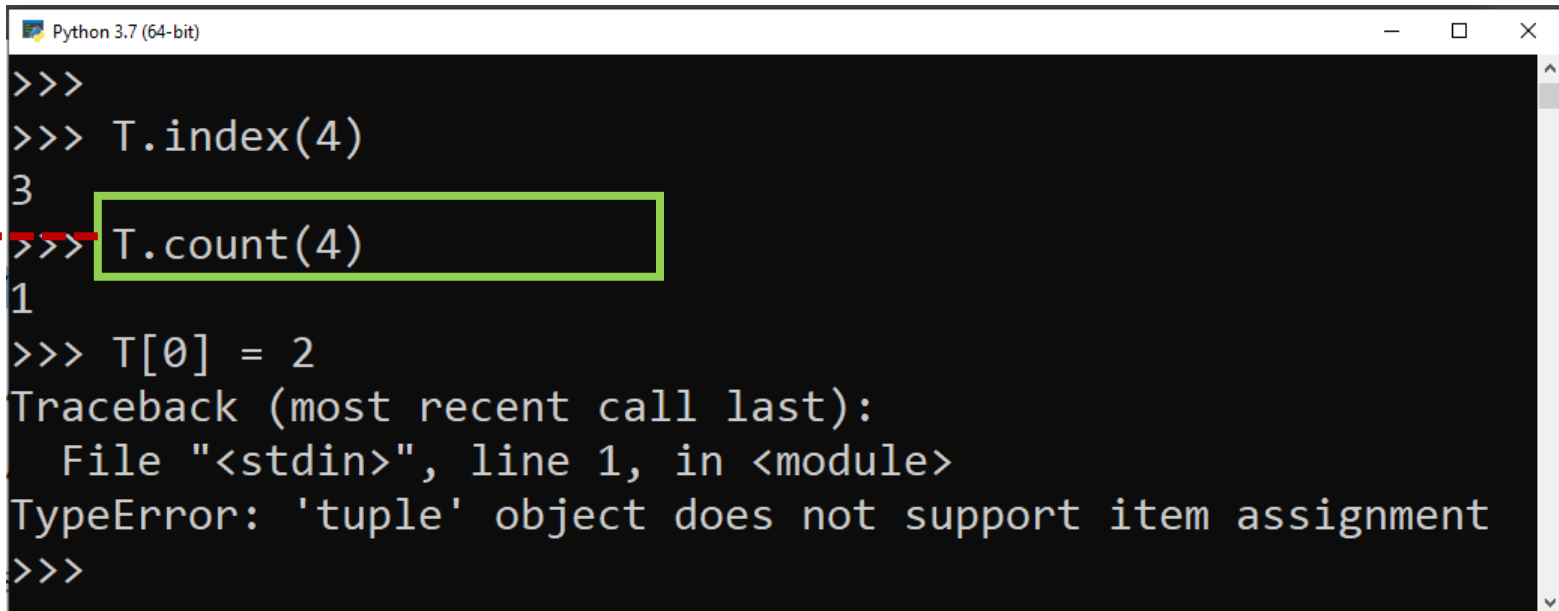
```
Python 3.7 (64-bit)
>>>
>>> T.index(4)
3
>>> T.count(4)
1
>>> T[0] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

A screenshot of a Python 3.7 (64-bit) terminal window. The window has a title bar with standard OS controls. The terminal shows a series of commands and their outputs. The first command is `T.index(4)`, which returns `3`. The second command is `T.count(4)`, which returns `1`. The third command is `T[0] = 2`, which results in a `TypeError: 'tuple' object does not support item assignment`. A red dashed line originates from the `T.index(4)` line and points to a text box on the right.

**Tuple methods: 4
appears at offset 3**

1.9.5 Tuples (con't)

- Tuples also have two type-specific callable methods



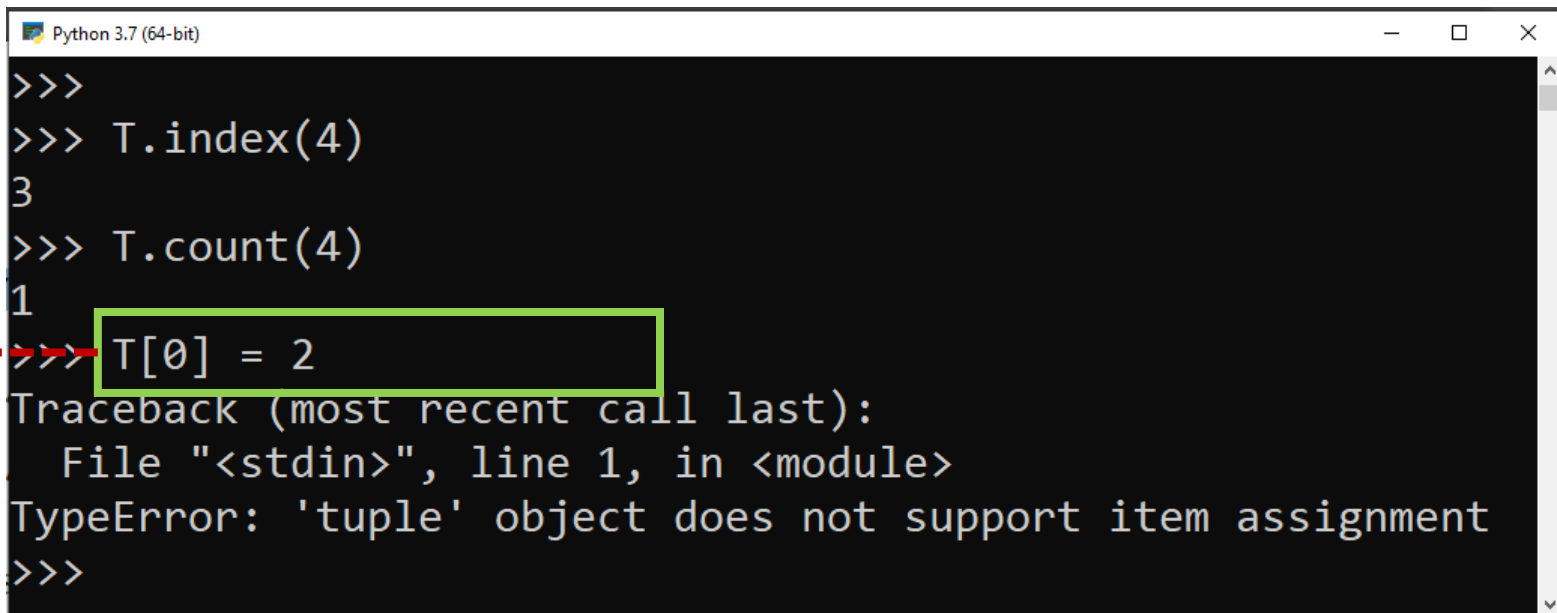
```
Python 3.7 (64-bit)
>>>
>>> T.index(4)
3
>>> T.count(4)
1
>>> T[0] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

The image shows a terminal window with a black background and white text. The window title is "Python 3.7 (64-bit)". The code entered is: `>>>`, `>>> T.index(4)`, `3`, `>>> T.count(4)`, `1`, `>>> T[0] = 2`. The output is: `3`, `1`, and a `TypeError: 'tuple' object does not support item assignment` traceback. A green rectangular box highlights the `T.count(4)` line. A red dashed line starts from the left side of this box, goes down, and then right to a red-bordered box containing the text "4 appears once".

4 appears once

1.9.5 Tuples (con't)

- Tuples also have two type-specific callable methods



```
Python 3.7 (64-bit)
>>>
>>> T.index(4)
3
>>> T.count(4)
1
>>> T[0] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

The image shows a terminal window titled "Python 3.7 (64-bit)". It contains several lines of code and output. The first three lines are interactive prompts: `>>>`, `>>> T.index(4)`, and `>>> T.count(4)`. The outputs are `3` and `1` respectively. The fourth line is `>>> T[0] = 2`, which is highlighted with a green rectangle. This line is crossed out with a red dashed line. Below this line, a traceback error is shown: `Traceback (most recent call last):`, `File "<stdin>", line 1, in <module>`, and `TypeError: 'tuple' object does not support item assignment`. The prompt `>>>` appears again at the bottom.

Tuples are immutable

1.9.5 Tuples (con't)

tuples support **mixed types** and **nesting**, but they don't grow and shrink because they are **immutable**

```
Python 3.7 (64-bit)
>>>
>>> T = ('spam', 3.0, [11, 22, 33])
>>> T[1]
3.0
>>> T[2][1]
22
>>> T.append(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>>
```

Mixed types

Why Tuples?

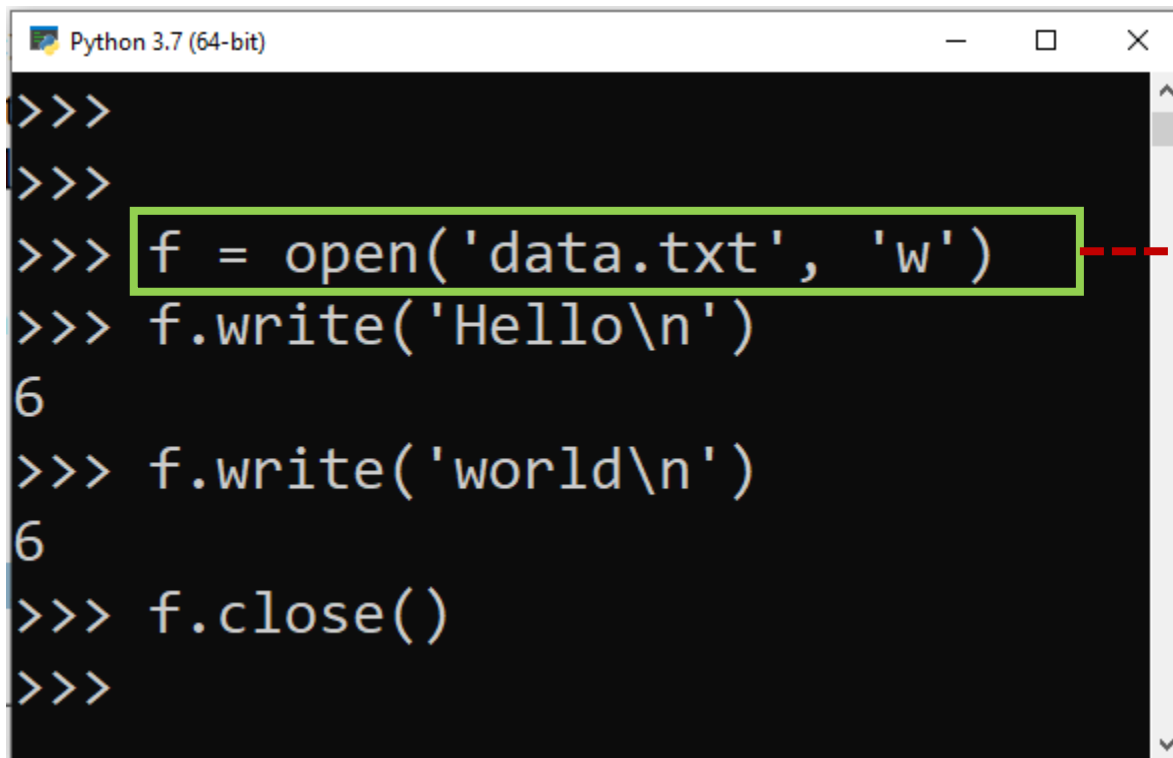
- If you pass a collection of objects around your program as a list, it can be **changed anywhere**;
- if you use a tuple, **it cannot**.

1.9.6 Files

- Files are a core type
- There is **no specific literal syntax** for creating them
- Call the built-in open function,
 - passing in an external filename and
 - a processing mode as strings

1.9.6 Files (con't)

- Write file

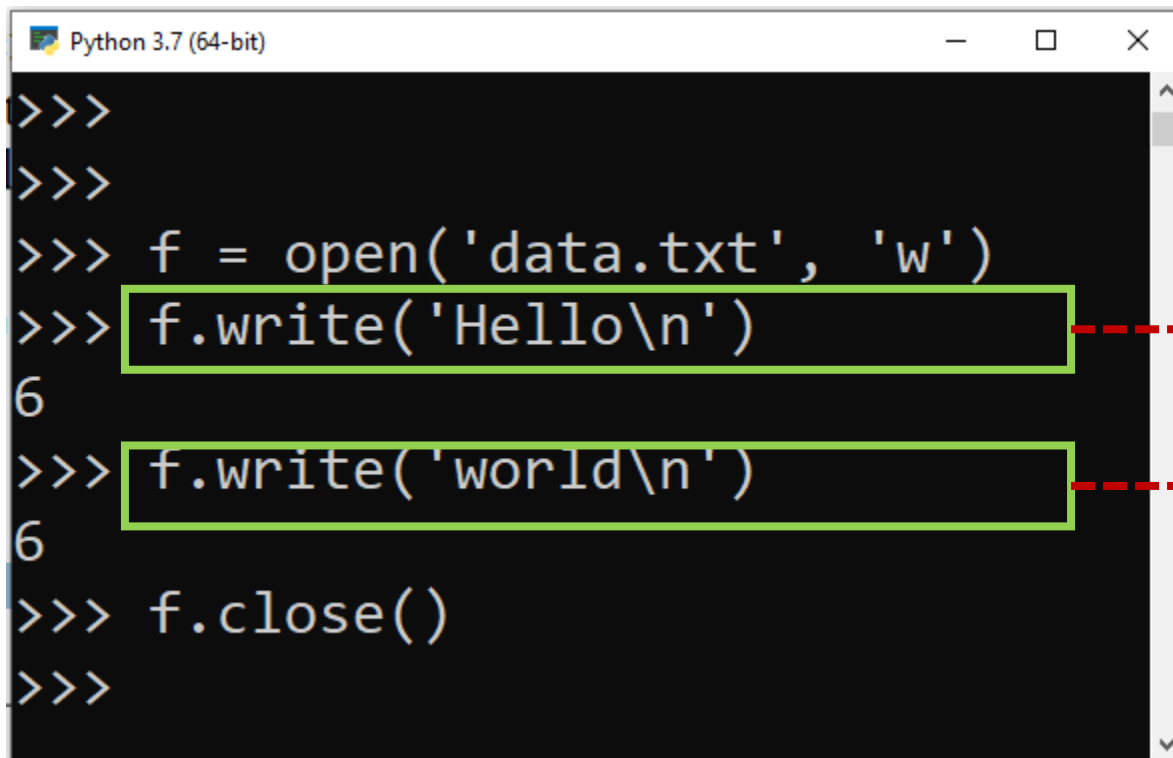


```
Python 3.7 (64-bit)
>>>
>>>
>>> f = open('data.txt', 'w')
>>> f.write('Hello\n')
6
>>> f.write('world\n')
6
>>> f.close()
>>>
```

**Make a new file
in output mode**

1.9.6 Files (con't)

- Write file



```
Python 3.7 (64-bit)
>>>
>>>
>>> f = open('data.txt', 'w')
>>> f.write('Hello\n')
6
>>> f.write('world\n')
6
>>> f.close()
>>>
```

**Write strings of
bytes to it**

1.9.6 Files (con't)

- Write file

```
Python 3.7 (64-bit)
>>>
>>>
>>> f = open('data.txt', 'w')
>>> f.write('Hello\n')
6
>>> f.write('world\n')
6
>>> f.close()
>>>
```

Returns number
of bytes written

1.9.6 Files (con't)

- Write file

```
Python 3.7 (64-bit)
>>>
>>>
>>> f = open('data.txt', 'w')
>>> f.write('Hello\n')
6
>>> f.write('world\n')
6
>>> f.close()
>>>
```

**Close to flush
output buffers
to disk**

1.9.6 Files (con't)

- Read file

```
Python 3.7 (64-bit)
>>>
>>> f = open('data.txt')
>>> text = f.read()
>>> text
'Hello\nworld\n'
>>> print(text)
Hello
world

>>> text.split()
['Hello', 'world']
>>>
```

**'r' is the default
processing
mode**

1.9.6 Files (con't)

- Read file

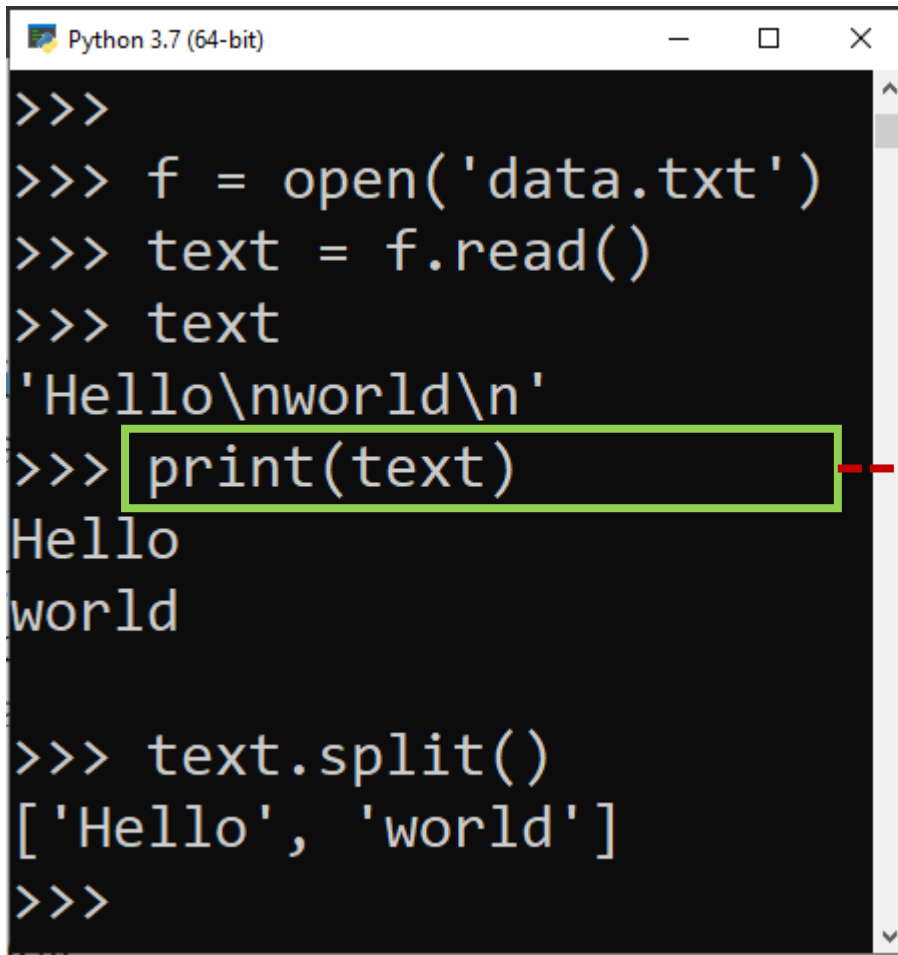
```
Python 3.7 (64-bit)
>>>
>>> f = open('data.txt')
>>> text = f.read()
>>> text
'Hello\nworld\n'
>>> print(text)
Hello
world

>>> text.split()
['Hello', 'world']
>>>
```

**Read entire file
into a string**

1.9.6 Files (con't)

- Read file



```
Python 3.7 (64-bit)
>>>
>>> f = open('data.txt')
>>> text = f.read()
>>> text
'Hello\nworld\n'
>>> print(text)
Hello
world

>>> text.split()
['Hello', 'world']
>>>
```

**print interprets
control characters**

1.9.6 Files (con't)

- Read file

```
Python 3.7 (64-bit)
>>>
>>> f = open('data.txt')
>>> text = f.read()
>>> text
'Hello\nworld\n'
>>> print(text)
Hello
world

>>> text.split()
['Hello', 'world']
>>>
```

File content is always
a **string**

Other File-Like Tools

- The *open* function is the workhorse for most file processing
- Advanced tasks:
 - pipes,
 - FIFOs,
 - sockets,
 - keyed-access files,
 - persistent object shelves,
 - descriptor-based files,
 - relational and object-oriented database interfaces,
 - and more

1.9.6 Other Core Types

- **Sets**
 - **unordered** collections of **unique** and
 - **immutable** objects
 - support the **usual mathematical set operations**
- **Decimal number**
- **Fraction number**
- **User-Defined Classes**

2. KÝ HIỆU VÀ CÁC PHÉP TOÁN

2. KÝ HIỆU VÀ CÁC PHÉP TOÁN

❖ Toán tử Python được chia thành 7 loại:

- Toán tử số học Python
- Toán tử so sánh (quan hệ) Python
- Toán tử gán trong Python
- Toán tử logic Python
- Toán tử Membership Python
- Toán tử xác thực Python
- Toán tử Bitwise trong Python

2.1 Toán tử số học

- Các toán tử số học Python này bao gồm các toán tử Python cho các phép toán cơ bản như:
 - Cộng, ký hiệu “+”
 - Trừ, ký hiệu “-”
 - Nhân, ký hiệu “*”
 - Chia, ký hiệu “/”
 - Chia lấy dư, ký hiệu “%”
 - Mũ, ký hiệu “**”
 - Chia làm tròn xuống, ký hiệu “//”

Toán tử Cộng (+)

- **Thêm giá trị** bằng cách **cộng** hai biến lại với nhau

```
>>> a = 10
```

```
>>> b = 20
```

```
>>> a + b
```

```
30
```


Toán tử Trừ (-)

- Trừ toán hạng bên phải khỏi toán hạng bên trái

```
>>> a = 10
```

```
>>> b = 20
```

```
>>> a - b
```

```
-10
```

Toán tử Nhân (*)

- Nhân hai giá trị lại với nhau

```
>>> a = 10
```

```
>>> b = 20
```

```
>>> a * b
```

```
200
```

Toán tử Chia (/)

- Chia hai giá trị cho nhau

```
>>> a = 10
```

```
>>> b = 20
```

```
>>> a / b
```

```
0.5
```

Toán tử Chia lấy dư (%)

- Chia toán hạng bên trái cho toán hạng bên phải và trả về phần dư.

```
>>> a = 10
>>> b = 20
>>> a % b
10
```

```
>>> a = 5
>>> b = 3
>>> a % b
2
```

```
>>> a = 5
>>> b = 4
>>> a % b
1
```

```
>>> a = 5
>>> b = 5
>>> a % b
0
```

Toán tử Mũ (**)

- Thực hiện phép tính lũy thừa trên các toán tử

```
>>> 2**2
```

```
4
```

```
>>> 2**3
```

```
8
```

```
>>> 2**4
```

```
16
```

```
>>> 3**2
```

```
9
```

```
>>> 3**3
```

```
27
```

```
>>> 3**4
```

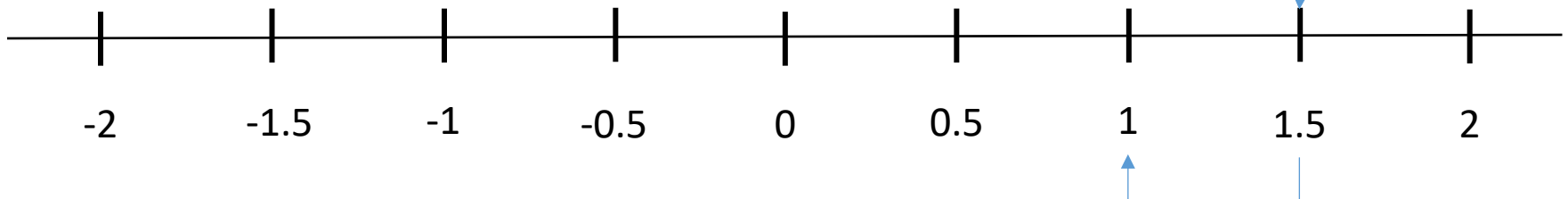
```
81
```

Toán tử Chia làm tròn (//)

- Sự phân chia các toán hạng trong đó kết quả là thương số, các chữ số sau dấu thập phân bị loại bỏ. Nhưng nếu một trong các toán hạng là số âm, kết quả sẽ được làm tròn, tức là, được làm tròn từ 0 (về phía âm vô cùng)

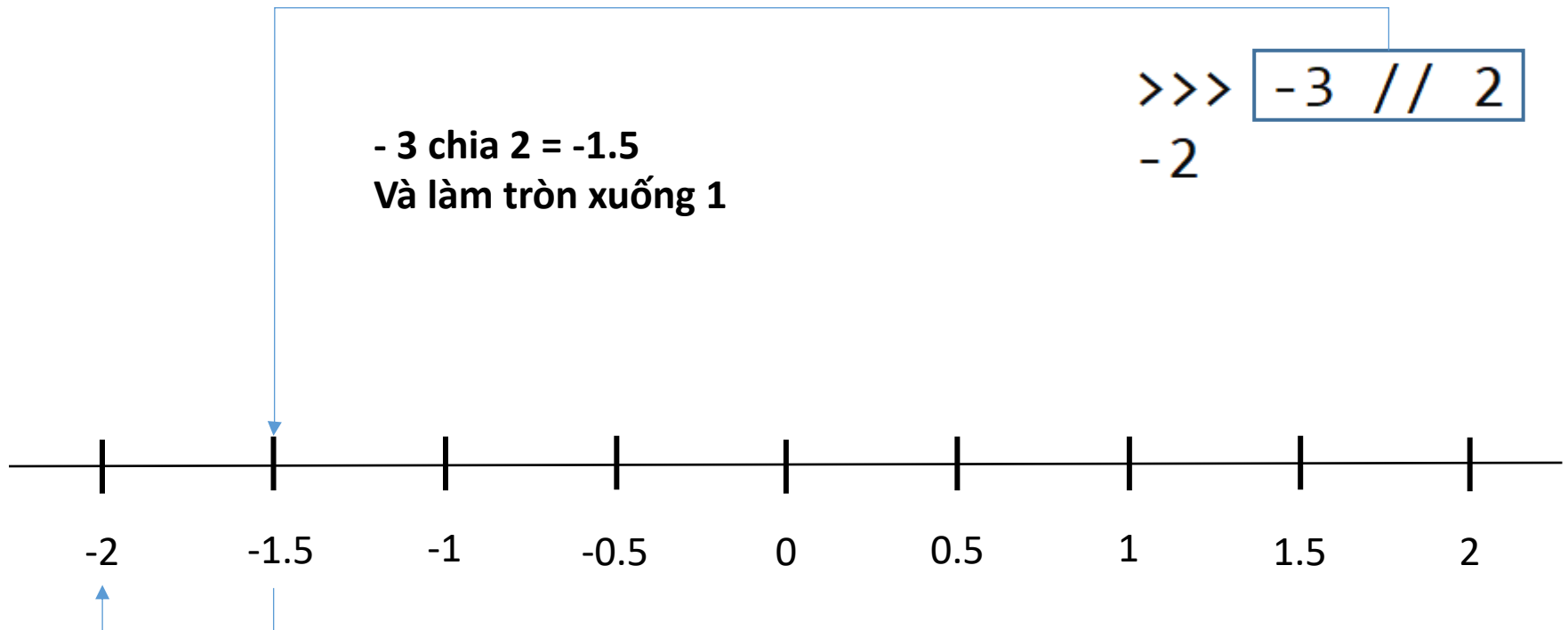
>>> 3 // 2
1

3 chia 2 = 1.5
Và làm tròn xuống 1



Toán tử Chia làm tròn (//)

- Sự phân chia các toán hạng trong đó kết quả là thương số, các chữ số sau dấu thập phân bị loại bỏ. Nhưng nếu một trong các toán hạng là số âm, kết quả sẽ được làm tròn, tức là, được làm tròn từ 0 (về phía âm vô cùng)



2.2 Toán tử so sánh (quan hệ)

- ❖ Dạng toán tử này dùng để so sánh các giá trị với nhau kết quả của nó sẽ trả về là True nếu đúng và False nếu sai. Và nó thường được dùng trong các câu lệnh điều kiện.
- ❖ Trong Python thì nó cũng tồn tại 6 dạng toán tử quan hệ cơ bản như sau:
 - Bằng (Equal), ký hiệu ==
 - Không bằng (Not Equal) !=
 - Nhỏ hơn (Small than), ký hiệu <
 - Lớn hơn (Greater than), ký hiệu >
 - Nhỏ hơn hoặc bằng, ký hiệu <=
 - Lớn hơn hoặc bằng, ký hiệu >=

2.2 Toán tử so sánh (quan hệ)

Operator	Description	Syntax	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	a==b	(1 == 2) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	a!=b	(1 != 2) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	a>b	(1 > 2) is not true.

2.2 Toán tử so sánh (quan hệ)

Operator	Description	Syntax	Example
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	$a < b$	$(1 < 2)$ is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	$a \geq b$	$(1 \geq 2)$ is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	$a \leq b$	$(1 \leq 2)$ is true.

Toán tử so sánh Bằng (==)

- So sánh giá trị của các đối số xem có bằng nhau hay không.
- Nếu bằng nhau thì kết quả trả về sẽ là True và ngược lại sẽ là False.

```
>>> 5 == 5
```

```
True
```

```
>>> 5 == 3
```

```
False
```

```
>>> type(5==5)
```

```
<class 'bool'>
```

Toán tử so sánh Không bằng (!=)

- So sánh giá trị của các đối số xem có khác nhau hay không.
- Nếu khác nhau thì kết quả trả về sẽ là True và ngược lại sẽ là False.

```
>>> 5 != 5
```

```
False
```

```
>>> 5 != 3
```

```
True
```

```
>>> type(5 != 3)
```

```
<class 'bool'>
```

Toán tử so sánh Nhỏ hơn (<)

- Dấu < đại diện cho phép toán nhỏ hơn, nếu đối số 1 nhỏ hơn đối số 2 thì kết quả sẽ trả về là True và ngược lại sẽ là False.

```
>>> 5 < 3
```

```
False
```

```
>>> 3 < 5
```

```
True
```

Toán tử so sánh Lớn hơn (>)

- Dấu > đại diện cho phép toán lớn hơn, nếu đối số 1 lớn hơn đối số 2 thì kết quả sẽ trả về là True và ngược lại sẽ là False.

```
>>> 5 > 3
```

```
True
```

```
>>> 3 > 5
```

```
False
```

Toán tử so sánh Nhỏ hơn hoặc bằng (<=)

- Dấu <= đại diện cho phép toán nhỏ hơn hoặc bằng, nếu đối số 1 nhỏ hơn hoặc bằng đối số 2 thì kết quả sẽ trả về là True và ngược lại sẽ là False.

```
>>> 3 < 4
```

```
True
```

```
>>> 3 <= 4
```

```
True
```

```
>>> 4 <= 3
```

```
False
```

```
>>>
```

Toán tử so sánh Lớn hơn hoặc bằng (\geq)

- Dấu \geq đại diện cho phép toán lớn hơn hoặc bằng, nếu đối số 1 lớn hơn hoặc bằng đối số 2 thì kết quả sẽ trả về là True và ngược lại sẽ là False.

```
>>> 3 >= 5
```

```
False
```

```
>>> 5 >= 5
```

```
True
```

```
>>> 5 >= 4
```

```
True
```


3.3 Toán tử gán

❖ Toán tử gán là toán tử dùng để **gán giá trị** của một đối tượng cho một đối tượng khác. Và trong Python thì nó cũng được thể hiện giống như các ngôn ngữ khác. Và dưới đây là 8 toán tử nằm trong dạng này mà Python hỗ trợ.

3.3 Toán tử gán

Operator	Description	Syntax	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	$a = b + c$	$c = a + b$ will assign value of $a + b$ into c
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	$a += b$	$c += a$ is equivalent to $c = c + a$
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	$a -= b$	$c -= a$ is equivalent to $c = c - a$
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	$a *= b$	$c *= a$ is equivalent to $c = c * a$

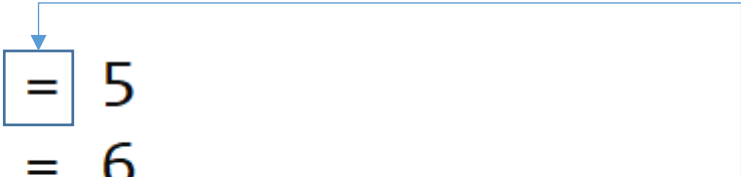
3.3 Toán tử gán

Operator	Description	Syntax	Example
<code>/=</code>	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	<code>a/=b</code>	<code>c /= a</code> is equivalent to <code>c = c / a</code>
<code>%=</code>	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	<code>a%b</code>	<code>c %= a</code> is equivalent to <code>c = c % a</code>
<code>**=</code>	Exponent AND assignment operator, Performs exponential (power) calculation on operators and assign value to the left operand	<code>a**=b</code>	<code>c **= a</code> is equivalent to <code>c = c ** a</code>
<code>//=</code>	Floor Division and assigns a value, Performs floor division on operators and assign value to the left operand	<code>a//=b</code>	<code>c //= a</code> is equivalent to <code>c = c // a</code>

Toán tử gán Bằng (=)

- Toán tử này dùng để gán giá trị của một đối tượng cho một giá trị.

```
>>> a = 5  
>>> b = 6  
>>> d = 'e'
```



Toán tử Gán =

Toán tử gán Cộng Bằng (+=)

- Toán tử này cộng rồi gán giá trị cho đối tượng

```
>>> a = 5
```

```
>>> a += 3
```

Toán tử Gán +=

```
>>> a
```

8

```
>>> a += 2
```

```
>>> a
```

10

```
>>> a += 4
```

```
>>> a
```

14

```
>>>
```

Toán tử gán Trừ Bằng (-=)

- Toán tử này Trừ rồi gán giá trị cho đối tượng

```
>>> a = 10
```

```
>>> a -= 1
```

```
>>> a
```

```
9
```

```
>>> a -= 5
```

```
>>> a
```

```
4
```

```
>>> a -= 4
```

```
>>> a
```

```
0
```

Toán tử Gán -=



Toán tử gán Nhân Bằng (*=)

- Toán tử này Nhân rồi gán giá trị cho đối tượng

```
>>> a = 3
```

```
>>> a *= 2
```

```
>>> a
```

```
6
```

```
>>> a *= 3
```

```
>>> a
```

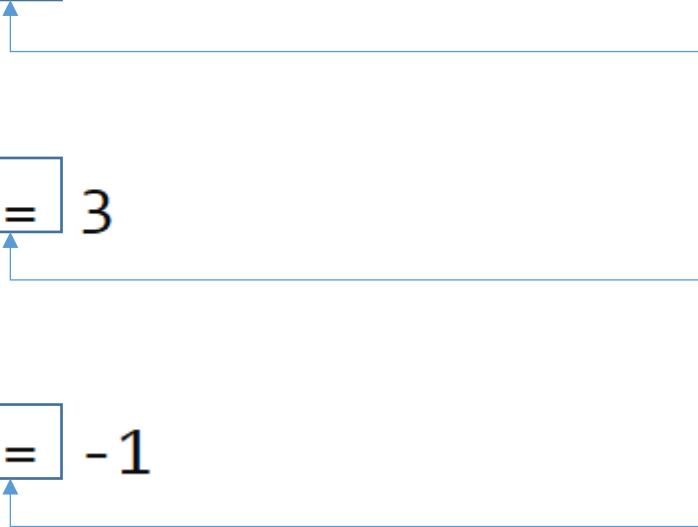
```
18
```

```
>>> a *= -1
```

```
>>> a
```

```
-18
```

Toán tử Gán *=



Toán tử gán Chia Bằng (/=)

- Toán tử này Chia rồi gán giá trị cho đối tượng

```
>>> a = 18
```

```
>>> a /= 2
```

Toán tử Gán /=

```
>>> a
```

```
9.0
```

```
>>> a /= 3
```

```
>>> a
```

```
3.0
```

```
>>> a /= -3
```

```
>>> a
```

```
-1.0
```


Toán tử gán Chia lấy dư bằng (//=)

- Toán tử này Chia lấy giá trị dư và gán giá trị dư đó cho đối tượng

```
>>> a = 15
```

```
>>> a %= 9
```

```
>>> a
```

```
6
```

```
>>> a %= 7
```

```
>>> a
```

```
6
```

```
>>> a %= 3
```

```
>>> a
```

```
0
```

Toán tử Gán //=

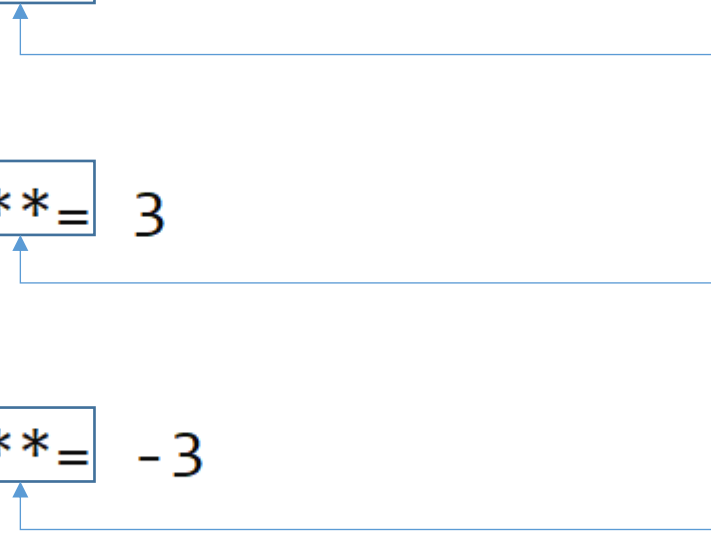


Toán tử gán Mũ bằng (**=)

- Toán tử này Lũy thừa rồi gán giá trị cho đối tượng

```
>>> a = 4
>>> a **= 2
>>> a
16
>>> a **= 3
>>> a
4096
>>> a **= -3
>>> a
1.4551915228366852e-11
>>>
```

Toán tử Gán **=



Toán tử gán Chia làm tròn bằng (//=)

- Toán tử này chia làm tròn rồi gán giá trị cho đối tượng

```
>>> a = 5
```

```
>>> a //= 3
```

Toán tử Gán //=

```
>>> a
```

1

```
>>> a //= 2
```

```
>>> a
```

0

3.4 Toán tử Logic

❖ Toán tử logic trong Python hoàn toàn giống như các ngôn ngữ khác. Nó gồm có 3 kiểu cơ bản như sau:

➤ and

➤ or

➤ not

Python Logical Operators:

Operator	Description	Syntax	Example
and	Called Logical AND operator. If both the operands are true then then condition becomes true.	a and b	(a and b) is true.
or	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.	a or b	(a or b) is true.
not	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	not a	not(a and b) is false.

Truth Table

A	B	A & B	A B	!A	!B
True	True	True	True	False	False
True	False	False	True	False	True
False	True	False	True	True	False
False	False	False	False	True	True

Toán tử logic AND

- Chỉ trả về True khi các điều kiện đều True

```
>>> a = False
```

```
>>> b = True
```

```
>>> c = True
```

```
>>> a and b
```

```
False
```

```
>>> b and c
```

```
True
```

```
>>> a and b and c
```

```
False
```

Toán tử Logic **and**



Toán tử logic OR

- Trả về **True** khi một trong các điều kiện là True

```
>>> a = False
```

```
>>> b = False
```

```
>>> c = True
```

```
>>> a or b
```

```
False
```

```
>>> a or c
```

```
True
```

```
>>> a or b or c
```

```
True
```

Toán tử Logic **or**

Toán tử logic NOT

- Trả về **True** khi điều kiện là False, và ngược lại

```
>>> a = True
```

```
>>> b = False
```

```
>>> not a and b
```

```
False
```

```
>>> a and not b
```

```
True
```

Toán tử Logic **not**



Python Bitwise Operators:

Bitwise operators act on bits and perform the **bit-by-bit** operations. These are used to operate on **binary** numbers.

Operator	Description	Syntax	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	$a \& b$	44&28 will give 12 which is 0000 1100
 	Binary OR Operator copies a bit if it exists in either operand.	$a b$	(44 28) will give 60 which is 0011 1100
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	$a \wedge b$	(44 ^ 28) will give 48 which is 0011 0000

Python Bitwise Operators:

Operator	Description	Syntax	Example
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	~a	(~44) will give -45 which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	a<<b	44 << 2 will give 176 which is 1011 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	a>>b	44 >> 2 will give 11 which is 0000 1011

Examples of Bitwise operators

Examples of Bitwise operators

a = 10

b = 4

Print bitwise **AND** operation

print(a & b)

Print bitwise **OR** operation

print(a | b)

Print bitwise **NOT** operation

print(~a)

print bitwise **XOR** operation

print(a ^ b)

print bitwise **right shift** operation

print(a >> 2)

print bitwise **left shift** operation

print(a << 2)

Examples of Bitwise operators

Examples of Bitwise operators

a = 3

b = 5

Print bitwise **AND** operation

print(a & b)

Print bitwise **OR** operation

print(a | b)

Print bitwise **NOT** operation

print(~a)

print bitwise **XOR** operation

print(a ^ b)

print bitwise **right shift** operation

print(a >> 2)

print bitwise **left shift** operation

print(a << 2)

Python Operators Precedence

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Complement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'

Python Operators Precedence

Operator	Description
<code>^ </code>	Bitwise exclusive `OR` and regular `OR`
<code><= < > >=</code>	Comparison operators
<code><> == !=</code>	Equality operators
<code>= %= /= //= -= += *= **=</code>	Assignment operators
<code>is is not</code>	Identity operators
<code>in not in</code>	Membership operators
<code>not or and</code>	Logical operators

Operator Precedence

This is used in an expression with more than one operator with different precedence to determine which operation to perform first.

Operator Associativity

If an expression contains two or more operators with the same precedence then Operator Associativity is used to determine. It can either be Left to Right or from Right to Left.

Example: Operator Precedence

```
expr = 10 + 20 * 30
```

```
print(expr)
```

```
# Precedence of 'or' & 'and'
```

```
name = "Alex"
```

```
age = 0
```

```
if name == "Alex" or name == "John" and age >= 2:
```

```
    print("Hello! Welcome.")
```

```
else:
```

```
    print("Good Bye!!")
```

Example: Operator Associativity

```
print(100 / 10 * 10)
```

```
print(5 - 2 + 3)
```

```
print(5 - (2 + 3))
```

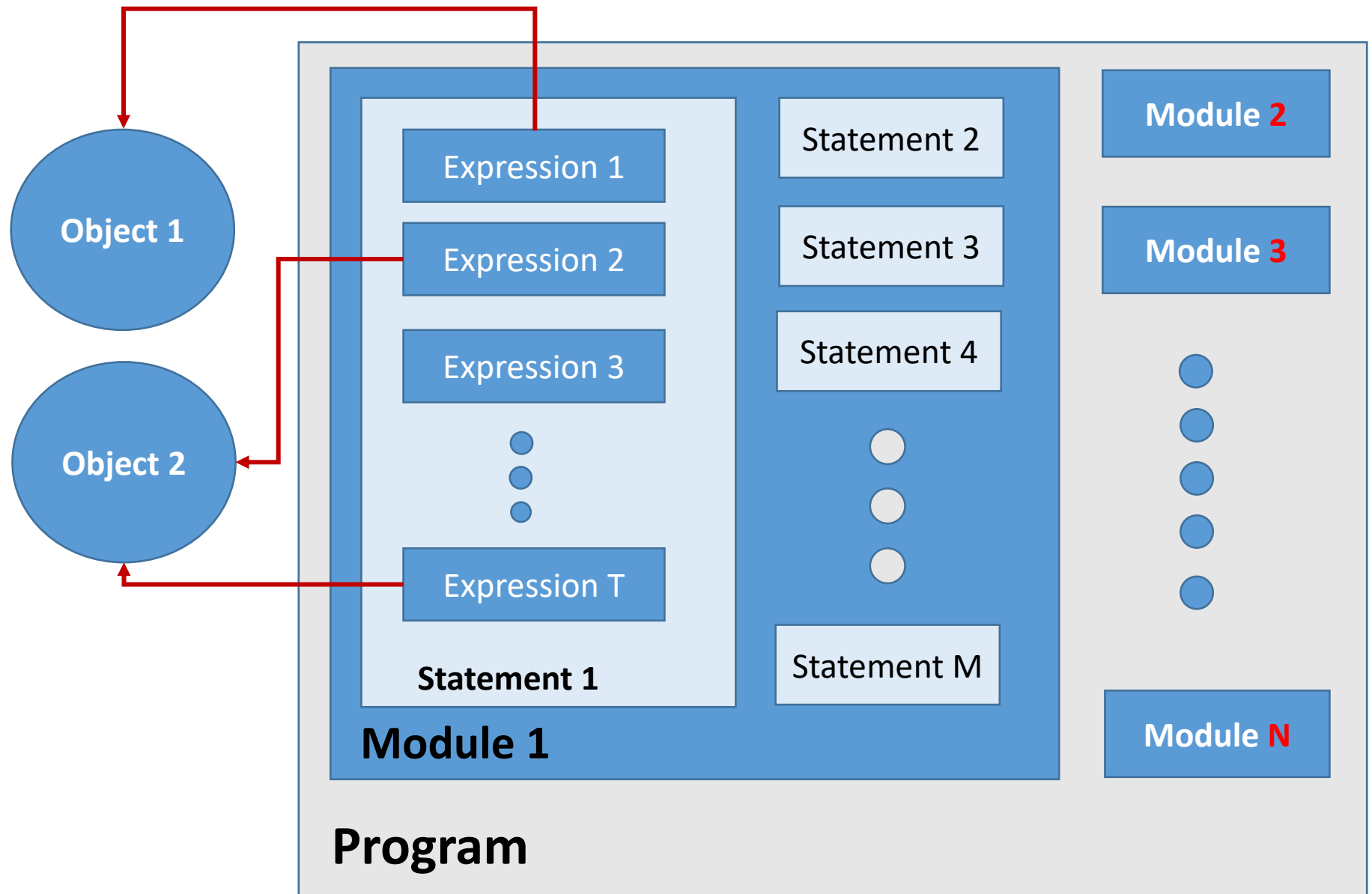
```
print(2 ** 3 ** 2)
```

3. CẤU TRÚC CƠ BẢN CỦA CHƯƠNG TRÌNH PYTHON

❖ Các chương trình Python có thể được phân tách thành các mô-đun, câu lệnh, biểu thức và đối tượng như sau:

- Chương trình bao gồm các mô-đun.
- Các mô-đun chứa các câu lệnh.
- Câu lệnh chứa biểu thức.
- Biểu thức và xử lý đối tượng.

Cấu trúc chương trình Python

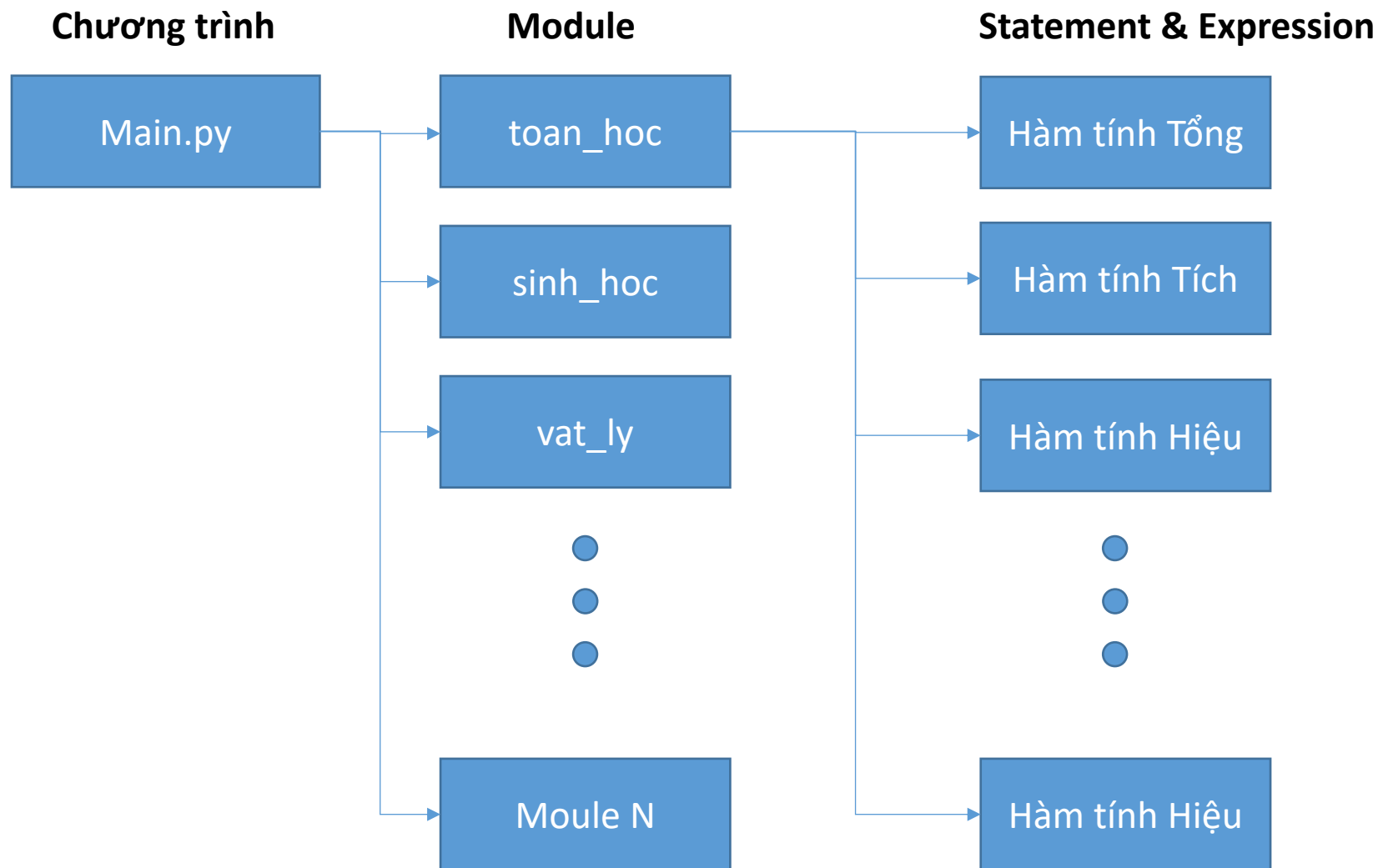


Module

- Module được sử dụng để phân loại code thành các phần nhỏ hơn liên quan với nhau.
- Module giúp tổ chức Python code một cách logic để giúp người dùng dễ dàng hiểu và sử dụng code đó hơn.
- Module là một **đối tượng** với các **thuộc tính** có thể đặt tên tùy ý và có thể **gắn kết** và **tham chiếu**
- Về cơ bản, một **Module là một file**, trong đó các lớp, hàm và biến được định nghĩa.
- Một Module cũng có thể bao gồm code có thể chạy.

Module (tt)

- Module trong Python là các **file** mà có các code tương tự nhau, hay có liên quan với nhau. Chúng có lợi thế sau:
 - **Khả năng tái sử dụng:** Module có thể được sử dụng ở trong phần Python code khác, do đó làm tăng tính tái sử dụng code.
 - **Khả năng phân loại:** Các kiểu thuộc tính tương tự nhau có thể được đặt trong một Module.



Ví dụ:

`import module toan_hoc`

`import toan_hoc`

`if __name__ == '__main__':`

```
toanHangA = input("Nhập A: ")
toanHangB = input("Nhập B: ")
ketQua = toan_hoc.cong(
    int(toanHangA),
    int(toanHangB))
print(ketQua)
```

docstring

Các khối lệnh

Lệnh

toan_hoc.py

`import sys`

`def cong(a, b):`

`"""tính tổng a và b"""`

`return a+b`

`def tru(a, b):`

`"""tính hiệu a và b"""`

`return a - b`

`def nhan(a, b):`

`"""nhân a và b"""`

`return a*b`

`def chia(a, b):`

`"""chia a và b"""`

`return a/b`

`def luythua(a, b):`

`"""a lũy thừa b"""`

`return a**b`

4. HÀM NHẬP XUẤT DỮ LIỆU

4.1 Nhập dữ liệu bằng hàm input() trong Python

- Hàm input() là một hàm có sẵn trong ngôn ngữ lập trình Python, hàm này cho phép người dùng **nhập vào dữ liệu từ bàn phím** sau đó **chuyển đổi thành một chuỗi** và trả về **nội dung đã nhập cho một biến**.
- Cú pháp: $\langle \text{tên biến} \rangle = \text{input}([\text{thông báo}])$
- Trong đó:
 - $\langle \text{tên biến} \rangle$: Là biến lưu trữ dữ liệu người dùng nhập vào từ bàn phím.
 - $[\text{thông báo}]$: Là chuỗi thông báo cho người dùng biết dữ liệu cần nhập, mặc định là chuỗi rỗng.

4.1 Nhập dữ liệu bằng hàm input() trong Python

```
>>> help(input)
```

Help on built-in function input in module builtins:

```
input(prompt=None, /)
```

Read a string from standard input. The trailing newline is stripped.
The prompt string, if given, is printed to standard output without a trailing newline before reading input.

If the user hits EOF (*nix: Ctrl-D, Windows: Ctrl-Z+Return), raise EOFError.
On *nix systems, readline is used if available.

Ví dụ 1:

- Ví dụ dưới đây sử dụng hàm **input()** để nhập dữ liệu họ tên và gán vào cho biến: **fullname** sau đó hiển thị dữ liệu vừa nhập ra màn hình như sau:

```
>>> fullname = input("Nhập Họ và Ten:")
Nhập Họ và Ten:Nguyen Van A
>>> fullname
'Nguyen Van A'
```

Ví dụ 2:

- Nhập vào 2 số và gán chúng vào 2 biến: a, b sau đó tính tổng hai số đó như sau:

```
a = input("Nhập số a: ")
b = input("Nhập số b: ")
# Chuyển dữ liệu về dạng số nguyên
a = int(a);
b = int(b);
# Tính tổng a + b và hiển thị kết quả
print("Kết quả:")
print(a + b)
```

Kết quả:

Nhập số a: 10

Nhập số b: 20

Kết quả:

30

4.2 Xuất dữ liệu bằng hàm print() trong Python

- Hàm print() trong Python được sử dụng để hiển thị dữ liệu ra màn hình. Các dữ liệu mà hàm print() có thể hiển thị đó là: kiểu số, kiểu chuỗi ký tự, kiểu list, kiểu dict, kiểu object....
- Khi xuất dữ liệu bằng hàm print() ta cũng có thể **format** thêm một số biến để đưa giá trị của biến đó ra màn hình.

Ví dụ 1:

Ví dụ sử dụng hàm print() hiển thị chuỗi và hiển thị số từ các biến ra màn hình như sau:

```
# Gan chuỗi và số vào biến
```

```
a = "Lập trình Python"
```

```
b = 22
```

```
# Hiện thị dữ liệu bằng print
```

```
print("Xin chào")
```

```
print(a)
```

```
print(b)
```

Xin chào

Lập trình Python

22

Ví dụ 2:

- Sử dụng hàm **print()** kết hợp hàm **format()** để định dạng và truyền dữ liệu vào chuỗi cần hiển thị ra màn hình như sau:

```
# Gán chuỗi và số vào biến
name = "Thanh"
age = 22
# Hiển thị dữ liệu bằng print và format
print("Xin chào, tôi là {0}, tôi {1} tôi".format(name, age))
```

Xin chào, tôi là Thanh, tôi 22 tuổi

Lưu ý: Các ngoặc nhọn {0}, {1}... được sử dụng để đặt dữ liệu cần hiển thị ra màn hình theo một thứ tự. Sau khi chương trình thực thi, các biến a, b sẽ hiển thị thay thế các {0}, {1}