# ADVANCED TOPICS

## Iterators trong Python

Trong bài này chúng ta sẽ tìm hiểu khái niệm **iterators trong Python**, đây là những đối tượng có những hàm hoặc phương thức hỗ trợ lặp riêng, thường là những kiểu dữ liệu dạng danh sách như mảng.

Mặc dù có thể sử dụng vòng lặp để lặp qua các phần tử của các iterators. Tuy nhiên, việc sử dụng những hàm hỗ trợ riêng của nó sẽ giúp ta lặp một cách an toàn hơn, điển hình nhất là việc ghi nhớ lại vị trí của phần đã lặp cuối cùng.

**Mục lục**

* [1. Iterators trong Python là gì?](https://freetuts.net/iterators-trong-python-3492.html#goto-h2-0)
  + [Lặp tuple với Iterators trong Python](https://freetuts.net/iterators-trong-python-3492.html#goto-h3-0)
  + [Lặp chuỗi bằng Iterators trong Python](https://freetuts.net/iterators-trong-python-3492.html#goto-h3-1)
  + [Lặp mảng bằng Iterators trong Python](https://freetuts.net/iterators-trong-python-3492.html#goto-h3-2)
* [2. lặp Iterators trong Python](https://freetuts.net/iterators-trong-python-3492.html#goto-h2-1)
* [3. Tự tạo ra Iterators trong Python](https://freetuts.net/iterators-trong-python-3492.html#goto-h2-2)
* [4. StopIteration trong Python](https://freetuts.net/iterators-trong-python-3492.html#goto-h2-3)

### Iterators trong Python là gì?

Trong Python, iterators là những đối tượng (object) được hỗ trợ những tính năng lặp riêng biệt, không cần phải sử dụng những vòng lặp thông thường như [vòng lặp for](https://freetuts.net/vong-lap-for-trong-python-1689.html) hay [vòng lặp while](https://freetuts.net/vong-lap-while-trong-python-685.html).

Một số đối tượng như [list](https://freetuts.net/kieu-du-lieu-string-trong-python-1711.html), [tuple](https://freetuts.net/kieu-du-lieu-string-trong-python-1711.html), [string](https://freetuts.net/kieu-du-lieu-string-trong-python-1711.html) là những **iterators**, bởi bạn có thể sử dụng hàm **iter()** để tạo chúng thành một iterators, sau đó sử dụng những hàm như **next()** để lặp qua từng phần tử.

Về mặt kỹ thuật thì một iterator object phải implement từ hai phương thức \_\_iter\_\_() và \_\_next\_\_(), được gọi chung là **iterator protocol**. Và để giúp lập trình viên sử dụng dễ dàng hơn thì hai build-in function ra đời, đó là **iter()** và **next()**.

#### Lặp tuple với Iterators trong Python

Hãy xem ví dụ dưới đây.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | # Danh sách dạng tuple  words = ("Xin", "chào", "các", "bạn")    # Chuyển nó thành Iterators  iword = iter(words)    # Sử dụng next để lấy từng phần tử  print(next(iword)) # Xin  print(next(iword)) # Chào  print(next(iword)) # Các  print(next(iword)) # Bạn |

Chạy lên kết quả sẽ như sau:



Hàm next sẽ trả về phần tử tiếp theo tính từ vị trí được đánh dấu là đã lặp cuối cùng. Chính vì vậy mình đã dễ dàng lấy ra 4 phần tử bằng bốn lệnh **next()**.

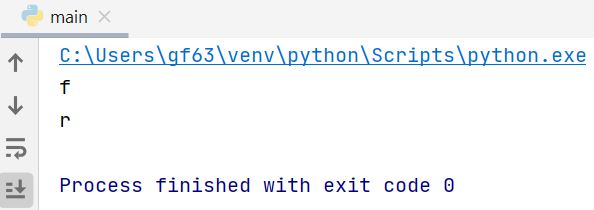
#### Lặp chuỗi bằng Iterators trong Python

Ta sẽ làm một ví dụ rất đơn giản, đó là in ra lần lượt từng kí tự của chuỗi "freetuts.net".

Đương nhiên bạn có thể sử dụng vòng lặp để làm bài này, nhưng cách đơn giản nhất là sử dụng iterators.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | # Chuỗi  domain = "freetuts.net"    # Chuyển thành Iterator  idomain = iter(domain)    # Lấy ký tự đầu tiên  print(next(idomain))    # Lấy ký tự thứ 2  print(next(idomain)) |

Kết quả:



#### Lặp mảng bằng Iterators trong Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | # Danh sách mảng  my\_list = [4, 7, 0, 3]    # Chuyển sang iterator  my\_iter = iter(my\_list)    # Lấy phần tử đầu tiên  print(next(my\_iter))    # Lấy phần tử tiếp theo (thứ 2)  print(next(my\_iter))    # Lấy phần tử tiếp theo (thứ 3)  print(my\_iter.\_\_next\_\_())    # Lấy phần tử tiếp theo (thứ 4)  print(my\_iter.\_\_next\_\_()) |

Trong ví dụ này mình có sử dụng my\_iter.\_\_next\_\_(), thực ra nó có công dụng giống với hàm next(my\_iter).

### lặp Iterators trong Python

Bạn có thể sử dụng vòng lặp for hoặc while để lặp qua các phần tử của đối tượng Iterators.

**Ví dụ vòng lặp for in 1**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | # Danh sách mảng  my\_list = [4, 7, 0, 3]    # Iterator  my\_iter = iter(my\_list)    # Lặp  for i in range(0, len(my\_list)):      print(next(my\_iter)) |

**Ví dụ vòng lặp for in 2**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | # Danh sách mảng  my\_list = [4, 7, 0, 3]    # Iterator  my\_iter = iter(my\_list)    # Lặp  for item in my\_iter:      print(item) |

**Ví dụ vòng lặp while**:

Bạn phải kết hợp với lênh **try .. except** để bắt lỗi khi lặp hết phần tử nhé.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | # Danh sách mảng  my\_list = [4, 7, 0, 3]    # Iterator  my\_iter = iter(my\_list)    # Lặp  while True:      try:          print(next(my\_iter))      except StopIteration:          break |

### Tự tạo ra Iterators trong Python

Để có thể tự tạo ra những đối tượng Iterators thì trong class bạn phải khai báo hai phương thức \_iter\_\_() và \_\_next\_\_().

* Phương thức \_\_iter\_\_() phải trả về chính đối tượng của lớp đó.
* Phương thức \_\_next\_\_() phải trả về phần tử tiếp theo trong dãy.

**Ví dụ**: Tạo một Iterator là danh sách các số chẵn lớn hơn 1.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | class MyNumbers:      def \_\_iter\_\_(self):          self.a = 2 # Số chẵn bắt đầu bằng 2          return self        def \_\_next\_\_(self):          x = self.a          self.a += 2 # Tăng lên 2 đơn vị          return x      mynumber = MyNumbers()  inumber = iter(mynumber)    print(next(inumber))  print(next(inumber))  print(next(inumber))  print(next(inumber))  print(next(inumber)) |

Kết quả:



### StopIteration trong Python

Bạn có thể sử dụng StopIteration để để thoát khỏi chương trình lặp.

Xét lại ví dụ ở trên, mình muốn MyNumbers chỉ in các số chẵn lớn hơn 1 và bé hơn 11.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | class MyNumbers:      def \_\_iter\_\_(self):          self.a = 2 # Số chẵn bắt đầu bằng 2          return self        def \_\_next\_\_(self):          x = self.a          self.a += 2 # Tăng lên 2 đơn vị          if (self.a < 11):              return x          else :              StopIteration    mynumber = MyNumbers()  inumber = iter(mynumber)    print(next(inumber))  print(next(inumber))  print(next(inumber))  print(next(inumber))  print(next(inumber)) # None |

Lệnh print cuối cùng nó trả về None vì kết quả đã vượt khỏi số 10.

Trên là cách sử dụng và tạo Iterators trong Python. Chúc bạn thành công nhé!

## Generator trong Python

Trong bài này chúng ta sẽ tìm hiểu về Generator trong Python, đây là cách giúp bạn tạo ra một đối tượng iterator cực kì dễ dàng. Bên cạn đó mình cũng phân tích giúp bạn hiểu được sự khác nhau giữa một hàm bình thường và một hàm generator.

**Mục lục**

* [1. Generators trong Python là gì?](https://freetuts.net/generator-trong-python-3494.html#goto-h2-0)
* [2. Cách tạo Generators trong Python](https://freetuts.net/generator-trong-python-3494.html#goto-h2-1)
* [3. Sự khác nhau giữa hàm generator và hàm bình thường](https://freetuts.net/generator-trong-python-3494.html#goto-h2-2)
* [4. Sử dụng vòng lặp trong Generator Python](https://freetuts.net/generator-trong-python-3494.html#goto-h2-3)
* [5. Sử dụng generator expression trong Python](https://freetuts.net/generator-trong-python-3494.html#goto-h2-4)
* [6. Tại sao nên dùng generator trong Python?](https://freetuts.net/generator-trong-python-3494.html#goto-h2-5)

### Generators trong Python là gì?

Generator là cách tạo ra một mô hình lặp [iterator](https://freetuts.net/iterators-trong-python-3492.html) trong Python, bằng cách sử dụng từ khóa **yield** để tạo ra những trình lặp một cách đơn giản và hiệu quả nhất.

Một hàm khi sử dụng yield thì bản thân nó đã tự kế thừa hai phương thức \_\_iter\_\_() và \_\_next\_\_() nên ta có thể sử dụng hàm next() mà không cần phải sử dụng hàm iter() để khởi tạo iterator.

### Cách tạo Generators trong Python

Việc tạo một generator trong Python khá là đơn giản, nó giống như một hàm bình thường, nhưng thay vì sử dụng lệnh return để trả về thì ta sử dụng lệnh [yield](https://freetuts.net/ref/yield-trong-python-dung-de-lam-gi-427.html).

Nếu một function (hàm) có chứa một hoặc nhiều lệnh yield thì nó là một generator. Một hàm có thể có một hoặc nhiều lệnh yield đặt tại nhiều vị trí khác nhau trong hàm.

Sự khác nhau giữa lệnh return và yield là trong khi return sẽ trả về một giá trị và kết thúc hàm, nhưng yield thì sẽ trả về nhưng vẫn giữ lại trạng thái của các biến, sau đó nếu được gọi tiếp thì nó sẽ tiếp tục xử lý ngay tại vị trí tạm dừng đó.

### Sự khác nhau giữa hàm generator và hàm bình thường

Dưới đây là một vài điểm khác nhau giữa hàm generator và hàm bình thường.

* Generator chứa một hoặc nhiều lệnh yield.
* Khi được gọi, hàm generator sẽ trả về một đối tượng iterator nhưng nó không thực thi liền.
* Hai phương thức \_\_iter\_\_() và \_\_next\_\_() được kế thừa tự động, vì vậy bạn có thể sử dụng hàm next() mà không cần dùng hàm iter() để tạo iterator .
* Trạng thái của các biến được lưu trữ lại giữa những lần gọi.
* Cuối cùng khi hàm kết thúc thì StopIteration sẽ bung ra cho lần gọi hàm tiếp theo, đây là lỗi cho thấy iterator đã duyệt đến phần tử cuối cùng.

Hãy xem ví dụ dưới đây để hiểu rõ hơn.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | # HỌC PYTHON TAI FREETUTS.NET  # AUTHOR: CƯỜNG NGUYỄN  def generateNumber():      n = 1      print("Lần gọi thứ nhất trả về n = ", n)      yield n        n += 1      print("Lần gọi thứ hai trả về n = ", n)      yield n        n += 1      print("Lần gọi thứ ba trả về n = ", n)      yield n    # Chương trình chính  num = generateNumber()    # Kiểm tra xe num là gì?  # => Nó là một generate object  # <generator object generateNumber at 0x000001CD0A939510>  print(num)    # Gọi đến generate => trả về yield đầu tiên  # Kết quả: 1  print(next(num))    # Gọi đến generate => trả về yield thứ hai  # Kết quả: 2  print(next(num))    # Gọi đến generate => trả về yield thứ ba  # Kết quả: 3  print(next(num)) |

Như bạn thấy, biến n bên trong hàm đã được nhớ sau mỗi lần gọi.

### Sử dụng vòng lặp trong Generator Python

Ở ví dụ trên mình chỉ giải thích cách hoạt động của generator chứ thực tế không ai làm như vậy.

Chúng ta thường sử dụng vòng lặp để tạo ra những generator.

Hãy xem ví dụ dưới đây, mình sẽ tạo ra một generator các số từ 1 đến 10.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | # Tạo generator  def generateNumber():      for i in range(1, 11):          yield i    # Lặp qua generator  num = generateNumber()  for n in num:      print(n) |

Kết quả như sau:



### Sử dụng generator expression trong Python

Ngoài những cách trên thì bạn có thể tạo ra generator bằng biểu thức expression.

Cách hoạt động của nó giống như [list comprehension](https://freetuts.net/list-comprehension-trong-python-3496.html), chỉ có điều một bên sử dụng dấu ngoặc vuông, một bên sử dụng dấu ngoặc nhọn.

Cách này chỉ phù hợp với những trường hợp có list data sẵn, và bạn muốn tạo ra một generator dựa trên list đó.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | # Tạo list  my\_list = [1, 3, 6, 10]    # Sử dụng expression để tạo generator  generator = (x\*\*2 for x in my\_list)    for item in generator:      print(item) |

**Kết quả**:



Ta có thể viết lại ví dụ trên bằng cách sử dụng vòng lặp như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | # Tạo list  my\_list = [1, 3, 6, 10]    # Sử dụng vòng lặp  def newList(my\_list):      for item in my\_list:          yield item    generator = newList(my\_list)    for item in generator:      print(item) |

### Tại sao nên dùng generator trong Python?

Lý do đơn giản và thuyết phục nhất là generator rất đơn giản và dễ thực hiện.

Generator triển khai dễ dàng và ngắn gọn hơn nhiều so với việc sử dụng trình lặp của iterator.

**Ví dụ**: Xây dựng một iterator tính lũy thừa của 2.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | class LuyThua2():      def \_\_init\_\_(self, max = 0):          # Thuộc tính lưu số lũy thừa hiện tại          self.n = 0          # Thuộc tính lưu số lũy thừa tối đa          self.max = max        def \_\_iter\_\_(self):          return self        def \_\_next\_\_(self):          if self.n > self.max:              raise StopIteration          else:              result = 2 \*\* self.n              self.n += 1              return result      l = LuyThua2(5)    print(next(l)) # n = 0  print(next(l)) # n = 1  print(next(l)) # n = 2  print(next(l)) # n = 3  print(next(l)) # n = 4  print(next(l)) # n = 5    print(next(l)) # Trả về lỗi StopIteration |

Nhưng nếu viết bằng Generator thì chương trình rất ngắn gọn như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | def LuyThua2(max = 0):      n = 0      while n <= max:          yield 2 \*\* n          n += 1    l = LuyThua2(5)  print(next(l)) # n = 1  print(next(l)) # n = 2  print(next(l)) # n = 3  print(next(l)) # n = 4  print(next(l)) # n = 5  print(next(l)) # n = 6  print(next(l)) # Lỗi StopIteration |

Ngoài ra, generator thân thiện hơn bởi nó chỉ tạo ra một mục tại một thời điểm gọi.

Trên là những chia sẻ cơ bản về cách sử dụng generator trong Python.

## List comprehension trong Python

Trong bài này chúng ta sẽ tìm hiểu cách sử dụng list comprehension trong Python, đây là cách tạo ra một list mới dựa vào một list có sẵn dựa vào một điều kiện nào đó.

**Mục lục**

* [1. List comprehension là gì?](https://freetuts.net/list-comprehension-trong-python-3496.html#goto-h2-0)
* [2. Ví dụ về list comprehension trong Python](https://freetuts.net/list-comprehension-trong-python-3496.html#goto-h2-1)

### List comprehension là gì?

List comprehension là cách tạo ra một list dữ liệu mới dựa vào dữ liệu cũ theo một điều kiện nào đó.

Vậy, công thức chung của list comprehension là:

|  |  |
| --- | --- |
| 1 | newlist = [expression for item in iterable if condition == True] |

Trong đó:

* expression chính là biểu thức trả về
* iterable là tập dữ liệu iterator
* condition là điều kiện trả về, phần tử nào có condition = True thì biểu thức expression tương ứng sẽ được trả về.

Giả sử mình có một mảng các số như sau:

|  |  |
| --- | --- |
| 1 | numbers = [1,3,86,4,54,54,45,65,6,57,67,87,82] |

Bây giờ mình muốn tạo một mảng mới từ mảng này, với điều kiện là chỉ lấy các số chẵn mà thôi.

**Cách thứ nhất**: Sử dụng vòng lặp bình thường.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | numbers = [1,3,86,4,54,54,45,65,6,57,67,87,82]    new\_numbers = []  for n in numbers:      if n % 2 == 0:          new\_numbers.append(n)    # kết quả: [86, 4, 54, 54, 6, 82]  print(new\_numbers) |

**Cách thứ hai**: Sử dụng list comprehension.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | numbers = [1,3,86,4,54,54,45,65,6,57,67,87,82]    new\_numbers = [x for x in numbers if x % 2 == 0]    # kết quả: [86, 4, 54, 54, 6, 82]  print(new\_numbers) |

Bạn có thể không cần phải truyền condition.

**Ví dụ**: Tạo mảng mới với giá trị gấp đôi mảng cũ.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | numbers = [1,3,5,7,9]    new\_numbers = [x\*2 for x in numbers]    # Kết quả: [2, 6, 10, 14, 18]  print(new\_numbers) |

### Ví dụ về list comprehension trong Python

Bây giờ mình sẽ làm thêm một vài ví dụ để bạn dễ hiểu hơn nhé.

**Ví dụ**: Tạo danh sách các số nguyên tố từ một danh sách có sẵn.

Để giải bài này thì ta cần tạo một hàm kiểm tra số nguyên tố trước, sau đó sẽ sử dụng nó trong phần **condition**.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | # Hàm kiểm tra số nguyên tố  def laSNT(n):      if n < 2: return False      elif n == 2: return True      elif n % 2 == 0: return False      else:          for i in range(3, n, 2):              if n % i == 0: return False      return True    # Dữ liệu  data = [1, 2, 3, 5, 45, 46, 5, 6, 76, 9, 11]    # Lấy danh sách các số nguyên tố trong data  data\_snt = [x for x in data if laSNT(x)]  print(data\_snt) |

Đây chính là cách sử dụng list comprehension để tạo ra danh sách mảng mới trong Python.

## Hàm closure trong Python

Trong bài này chúng ta sẽ tìm hiểu hàm closure trong Python, cách tạo ra một hàm closure, các ví dụ cách sử dụng, cũng như lý do tại sao nên dùng closure Python.

Trước khi tìm hiểu về closure thì bạn phải hiểu hai khái niệm, đó là hàm lồng nhau (nested function) và biến không cục bộ (non-local variable). Nào, hãy bắt đầu ngay nhé.

**Mục lục**

* [1. Nested function là gì?](https://freetuts.net/closure-trong-python-3497.html#goto-h2-0)
* [2. Cách khai báo closure function trong Python](https://freetuts.net/closure-trong-python-3497.html#goto-h2-1)
* [3. Tại sao nên sử dụng closure function trong Python](https://freetuts.net/closure-trong-python-3497.html#goto-h2-2)

### Nested function là gì?

Nested function Python hay còn gọi là **hàm lồng nhau** trong Python.

Một hàm được định nghĩa bên trong một hàm khác thì ta gọi đó là nested function. Các hàm nested function có thể sử dụng biến trong phạm vi hàm cha của nó nhưng không được thay đổi giá trị của chúng.

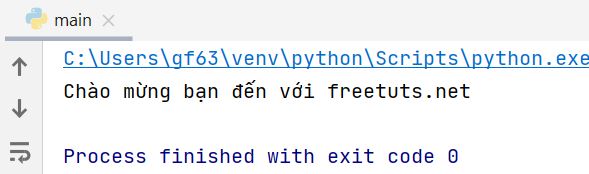
Non-local variable (biến không cục bộ) có thể được tạm hiểu là một biến được khai báo trong một hàm A, và nó cũng có thể được sử dụng bên trong những hàm nested function khai báo bên trong hàm A đó.

Bạn **không được thay đổi giá trị**của biến không cục bộ.

Ví dụ dưới đây thì hàm **printer** là một nested function và chúng truy cập vào một biến không cục bộ tên là **msg**.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | def sayHi(msg):        # Phạm vi code bên ngoài        def printer():          # Đây là nested function          print(msg)        printer()    # Thực hiện hàm cha  # Trong thâm hàm cha gọi đến một nested fucntion  # có tên là printer  sayHi("Chào mừng bạn đến với freetuts.net") |

Kết quả như sau:



Như bạn thấy, nested function tên là **printer**đã sử dụng biến của hàm cha của nó tên là **msg**. Biến **msg**nó có phạm vi bên trong hàm **sayHi**, và vì **printer**là hàm được khai báo trong hàm **sayHi**nên nó có thể sử dụng biến **msg**.

Dưới đây là một ví dụ mình đa cố tình thay đổi giá trị cho biến không cục bộ, đây là một cú pháp sai.

|  |  |
| --- | --- |
| 1  2  3  4  5 | def createNum():      n = 1        def plusOne() :          n += 1 # Sai vì đã thay đổi giá trị cho biến n |

### Cách khai báo closure function trong Python

Xét lại ví dụ ở trên, chuyện gì sẽ xảy ra nếu ta đặt lệnh gọi đến hàm printer() bên trong lệnh **return**? Điều này có nghĩa là ta đang trả về một hàm printer chứ không phải là gọi thực thi nó.

Lúc này việc sử dụng hàm **sayHi**sẽ qua hai công đoạn:

* Thứ nhất: Gọi hàm sayHi, hàm này sẽ trả về hàm printer
* Thứ hai: Gọi đến hàm printer bằng cách gọi đến biến lưu trữ giá trị trả về từ hàm sayHi.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | def sayHi(msg):      # Phạm vi code bên ngoài        def printer():          # Đây là nested function          print(msg)        return printer    # Gọi đến hàm sayHi  s = sayHi("Welcome to freetuts.net")    # In thử xem nó là cái gì  print(s)  # Kết quả: <function sayHi.<locals>.printer ...>    # Vì nó là một hàm printer nên ta gọi thêm một lần nữa  s() |

Bạn có nhận thấy điều bất thường không? Khi gọi đến hàm **sayHi**thì biến **msg**đã được gán giá trị là "Welcome to freetuts.net", và sau đó biến đó **vẫn được lưu trữ giá trị** cho đến khi mình gọi đến hàm printer bằng lệnh s().

Cách viết như thế này ta gọi là **closure function**. Vậy closure function hàm mà giá trị trả về một hàm khác.

### Tại sao nên sử dụng closure function trong Python

Từ ví dụ trên thì ta có kết luận rằng các biến không cục bộ trong closure function sẽ **được ghi nhớ**, vì vậy ta có thể tận dụng nó để làm những công việc sau đây.

**Đầu tiên** là có thể thay thế trong lập trình hướng đối tượng, thay vì phải tạo class dài dòng thì bạn có thể dùng closure.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | # Chương trình sẽ tính n ^ x  def soMu(n):      def cal(x):          return n \*\* x      return cal    # Gọi đến hàm soMu  # Hàm này trả về hàm cal  mu = soMu(2)    # Tính 2 mũ 5  print(mu(5))    # Tính 2 mũ 8  print(mu(8)) |

Ta có thể viết bằng class như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class soMu:      def \_\_init\_\_(self, n):          self.n = n        def cal(self, x):          return self.n \*\* x    mu = soMu(2)    print(mu.cal(5)) # 2 ^ 5  print(mu.cal(8)) # 2 ^ 8 |

Trên là cách sử dụng closure trong Python, hy vọng qua bài viết này sẽ giúp bạn hiểu khái niệm closure python là gì, cách khai báo nó như thế nào, cũng như nắm vững được khái niệm **nested function** và **non-local variable**.

## Higher Order Functions trong Python

Trong bài này chúng ta sẽ tìm hiểu Higher Order Functions trong Python, qua đó mình cũng nói qua cách sử dụng Decorators, một cách sử dụng khá phổ biến của Higher Order Functions.

Trước khi tìm hiểu thì mình muốn nhắc các bạn rằng bản thân hàm (function) trong Python là một object, được kế thừa tự động một số phương thức như \_\_call\_\_, cũng như bạn có thể gán nó vào một biến và gọi lại sau đó.

**Mục lục**

* [1. Higher order function trong Python là gì?](https://freetuts.net/higher-order-functions-3500.html#goto-h2-0)
* [2. Các ví dụ về higher order function trong Python](https://freetuts.net/higher-order-functions-3500.html#goto-h2-1)
  + [Functions là objects](https://freetuts.net/higher-order-functions-3500.html#goto-h3-0)
  + [Function là một tham số](https://freetuts.net/higher-order-functions-3500.html#goto-h3-1)
  + [Return về một function](https://freetuts.net/higher-order-functions-3500.html#goto-h3-2)
* [3. Điểm qua một chút về Decorator](https://freetuts.net/higher-order-functions-3500.html#goto-h2-2)

### Higher order function trong Python là gì?

Trong Python, một hàm được gọi là higher order function (hàm bậc cao) nếu hàm đó có tham số truyền vào là một hàm khác, và bản thân hàm này cũng có thể return về một function (ta gọi đó là [*closure function*](https://freetuts.net/closure-trong-python-3497.html)).

Sau đây là một số đặc điểm:

* Một function là một instance của một object
* Bạn có thể lưu trữ function trong một biến
* Bạn có thể truyền một function vào tham số của một function khác
* Bạn có thể return về một function
* Bạn có thể lưu trữ function trong các kiểu dữ liệu dạng cấu trúc như list, hash tables.

Ta hãy thử **print**xem **type**của một function là gì nhé.

|  |  |
| --- | --- |
| 1  2  3  4  5 | def abc():      pass    t = abc()  print(type(t)) |

Kết quả:

|  |  |
| --- | --- |
| 1 | <class 'NoneType'> |

Như vậy biến t chính là một class.

### Các ví dụ về higher order function trong Python

Bây giờ ta sẽ làm một vài ví dụ về những đặc điểm của funciton cấp cao trong Python nhé.

#### Functions là objects

Function là object nên bạn có thể lưu trữ nó trong một biến, sau đó gọi thông qua biến đó.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | def sayHi(msg):      print(msg)    # Gán function vào biến  say = sayHi    # Gọi function  say("freetuts.net") |

Kết quả:



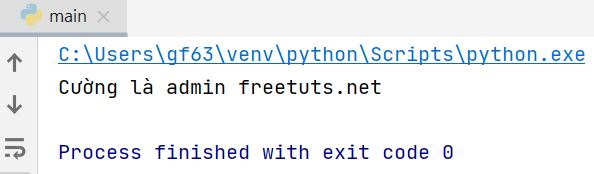
Như trong ví dụ này thì bạn thấy mình đã lưu trữ function **sayHi**vào biến **say**, sau đó gọi đến function bằng cú pháp **say()**.

#### Function là một tham số

Ta có thể truyền function vào như là một tham số, bên trong hàm cha sẽ gọi để kích hoạt function tham số đó.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | # Hàm cha có tham số getMsg là một function  def sayHi(getMsg, msg):      # Gọi đến hàm getMsg, và truyền tham số cho nó      print(getMsg(msg))    # Hàm getMsg có một tham số truyền vào  def getMsg(msg):      return msg + " freetuts.net"    # Chương trình chính  msg = getMsg  sayHi(msg, "Cường là admin") |

Kết quả:

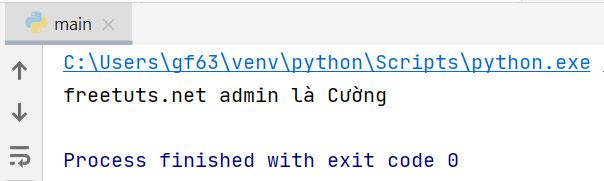


#### Return về một function

Dạng này ta gọi là closure function nhé các bạn. Nếu bạn muốn tìm hiểu sâu hơn thì hãy đọc bài đó trong series Python nâng cao này nhé.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def sayHi(domain):      def getMsg(msg):          return domain + msg      return getMsg    hi = sayHi("freetuts.net")  print(hi(" admin là Cường")) |

Kết quả:



### Điểm qua một chút về Decorator

Decorator là kỹ thuật dùng higher order function để trang trí thêm nội dung cho một function có sẵn.

Hãy xem ví dụ dưới đây.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | def enclose(fun):        def wrapper():            print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")          fun()          print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")        return wrapper    def myfun():      print("myfun")    enc = enclose(myfun)  enc() |

Trong ví dụ này mình đã sử dụng hàm enclose là một decorator, bởi vì mình dùng nó để gọi đến hàm fun(), và đặt hàm func() bên trong hai hàng star.

Như vậy khi chạy thì hai hàng star được trang trí thêm cho hàm func (in nội dung my fun).

Xem bài viết [decorator trong Python](https://freetuts.net/decorator-trong-python-3501.html) để hiểu thêm nhé.

Trên là bài hướng dẫn cách sử dụng hàm bậc cao "higher order function trong Python", hy vọng qua bài này sẽ giúp bạn hiểu được cách truyền tham số vào hàm là một tham số khác. Chúc bạn thành công.

## Decorator trong Python

Trong bài này chúng ta sẽ tìm hiểu cách dùng Decorator trong Python, dựa vào cấu trúc của nested function ta có thể tạo ra những decorator một cách dễ dàng.

Các hàm trong Python là first-class (lớp đầu tiên), điều này có nghĩa là các hàm sẽ có trạng thái bình đẳng với những đối tượng object khác trong Python. Điều này có nghĩa là các hàm có thể được gán cho biến, được truyền vào như một tham số, được lưu trữ trong các collection.

**Mục lục**

* [1. Python decorator là gì?](https://freetuts.net/decorator-trong-python-3501.html#goto-h2-0)
* [2 Tạo decorator trong Pyhton bằng dấu @](https://freetuts.net/decorator-trong-python-3501.html#goto-h2-1)
* [3. Python decorator với hàm có tham số](https://freetuts.net/decorator-trong-python-3501.html#goto-h2-2)
* [4. Multi decorator trong Python](https://freetuts.net/decorator-trong-python-3501.html#goto-h2-3)
* [5. Decorator với class trong Python](https://freetuts.net/decorator-trong-python-3501.html#goto-h2-4)

### Python decorator là gì?

Python decorator là cách thay đổi hành vi (behavior) của một object [callable](https://freetuts.net/ref/callable-trong-python-428.html) mà không cần can thiệp và chỉnh sửa object đó. Decorator là một hàm có nhiệm vụ bao bọc một hàm khác (ta tạm gọi là hàm B), hàm B sẽ được gọi bên trong thân của hàm decorator, và đương nhiên lúc này hàm decorator có thể bổ sung những đoạn code phía trên và phía dưới vị trí gọi hàm B.

**Hãy lấy một ví dụ cho dễ hiểu**: Giả sử bạn viết một hàm in ra các số từ 1 đến 10 như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5 | def showNumbers():      for i in range( 1, 11):          print(i, end=' ')    showNumbers() |

Nhưng hôm sau bạn muốn in thêm một dòng dấu sao ở phía trên và phía dưới của các số đó nữa thì làm thế nào? Cách đơn giản nhất là sửa lại hàm đó.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | def showNumbers():      print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')      for i in range( 1, 11):          print(i, end=' ')      print('')      print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')    showNumbers() |

Nhưng cách làm này sẽ không hiệu quả nếu function showNumbers mình đã sử dụng ở rất nhiều nơi, nếu sửa trực tiếp như vậy sẽ làm ảnh hưởng đến những vị trí đó.

Để giải quyết thì mình sẽ kết hợp với [higher order function](https://freetuts.net/higher-order-functions-3500.html) để tạo ra decorator.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | # Đây là hàm decorator  def decoratorFunc(func):      # Đây là hàm decorator gọi,      # ta gọi là extend decorated function      def addStar():          print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")          func()          print('')          print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")        return addStar    def showNumbers():      for i in range( 1, 11):          print(i, end=' ')      add\_star = decoratorFunc(showNumbers)  add\_star() |

Trong ví dụ này thì hàm decoratorFunc là một **decorator**, bên trong được gọi đến hàm mở rộng addStar dùng để trang trí thêm mấu dấu sao.

### Tạo decorator trong Pyhton bằng dấu @

Python cho phép bạn sử dụng dấu @ để đánh dấu một phương thức sẽ được trang trí (decorator) bằng một phương thức khác.

Quay lại ví dụ trên thì mình sẽ viết lại chương trình decorator như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | def decoratorFunc(func):      def addStar():          print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")          func()          print('')          print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")        return addStar    @decoratorFunc  def showNumbers():      for i in range( 1, 11):          print(i, end=' ')    # Gọi kiểu này đơn giản hơn  showNumbers() |

Kết quả trả về là như nhau:



Kể từ bây giờ chúng ta nên sử dụng cách này nhé các bạn, bởi trông chuyên nghiệp và ảo diệu hơn rất nhiều, một người không rành về Python sẽ không thể hiểu là bạn đang code cái gì :3

### Python decorator với hàm có tham số

Trường hợp hàm showNumbers có tham số thì cách viết như thế nào nhỉ?

Giả sử mình fix lại hàm đó với một tham số **n** truyền vào, và nội dung sẽ in ra dãy số từ 1 đến **n**.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | def decoratorFunc(func):      def addStar(n):          print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")          func(n)          print('')          print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")        return addStar    @decoratorFunc  def showNumbers(n):      for i in range( 1, n + 1):          print(i, end=' ')    # Gọi có tham số truyền vào  showNumbers(20) |

Các bạn chú ý nhé, đoạn code trên mình có truyền vào tham số n = 20 nên có hơi chút khác biệt so với các ví dụ ở trên.

### Multi decorator trong Python

Bạn có thể áp dụng nhiều decorator cho một function.

**Ví dụ**: Giả sử mình có một hàm

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | def addStrongTag(func):      def add(msg):          print('<strong>')          func(msg)          print('</strong>')      return add    def addSpanTag(func):      def add(msg):          print('<span>')          func(msg)          print('</span>')      return add    @addStrongTag  @addSpanTag  def showMessage(msg):      print(msg)    showMessage("Freetuts.net") |

Kết quả:



### Decorator với class trong Python

Bạn có thể tạo decorator bằng một class với điều kiện trong class đó phải khai báo phương thức \_\_call\_\_ nhé.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class addTag():        def \_\_init\_\_(self, func):          self.func = func        def \_\_call\_\_(self, msg):          print('<strong>')          self.func(msg)          print('</strong>')    @addTag  def showMessage(msg):      print(msg)    showMessage("freetuts.net") |

Kết quả:

|  |  |
| --- | --- |
| 1  2  3 | <strong>  freetuts.net  </strong> |

**Kết luận**: Qua bài này ta thấy việc sử dụng decorator trong Python thực sự rất hữu ích. Có nhiều trường hợp trong thực tế có thể áp dụng nó như: Chương trình quản lý đăng nhập, quản lý việc hiển thị nội dung ra bên ngoài một trang web ...

Cuối cùng, chúc bạn nắm vững được kỹ thuật decorator này nhé.

## Xử lý file và thư mục trong Python

Trong bài viết này ta sẽ tìm hiểu về các hoạt động với file trong Python như: mở file, đọc file, ghi file, đóng file, ... và các phương thức xử lý file khác.

**Mục lục**

* [1. Sơ lược về xử lý file trong Python](https://freetuts.net/xu-ly-file-trong-python-1714.html#goto-h2-0)
* [2. Mở file với open() trong Python?](https://freetuts.net/xu-ly-file-trong-python-1714.html#goto-h2-1)
* [3. Đóng kế nối với close() trong Python](https://freetuts.net/xu-ly-file-trong-python-1714.html#goto-h2-2)
* [4. Ghi file trong Python](https://freetuts.net/xu-ly-file-trong-python-1714.html#goto-h2-3)
* [5. Đọc dữ liệu của file trong Python](https://freetuts.net/xu-ly-file-trong-python-1714.html#goto-h2-4)
* [6. Lời kết](https://freetuts.net/xu-ly-file-trong-python-1714.html#goto-h2-5)

### Sơ lược về xử lý file trong Python

File là một tệp được người dùng máy tính sử dụng để lưu trữ dữ liệu của họ, nó sẽ được lưu trữ trên ổ đĩa vĩnh viễn trừ khi ổ đĩa bị hỏng hoặc bạn xóa nó.

Khi làm việc với file thì chúng ta sẽ có những hành động chính như sau:

* Mở, tạo file
* Thay đổi dữ liệu của file
* Xóa file
* Di chuyển file
* Copy file
* ...

Đối với một phiên làm việc với file trong Python thì sẽ có ba bước chính như sau:

* Mở file, nếu file cần mở không tồn tại thì tạo file mới.
* Thay đổi dữ liệu của file
* Đóng file

Bây giờ chúng ta sẽ tìm hiểu chi tiết hơn từng thao tác nhé.

### Mở file với open() trong Python?

Python cung cấp cho chúng ta một hàm có tên là open(), hàm này có cú pháp như sau:

|  |  |
| --- | --- |
| 1 | f = open("location\_file", "mode") |

Trong đó location\_file là đường dẫn trỏ đến file cần mở, còn mode là chế độ mở file. Bảng dưới đây là danh sách các chế độ mở file thường gặp.

|  |  |
| --- | --- |
| **Python File Modes** | |
| **Mode** | **Description** |
| 'r' | Mở file ở chế độ đọc (mặc định) |
| 'w' | Mở file ở chế độ ghi, tạo file mới nếu file không tồn tại, làm sạch file nếu nó đã tồn tại |
| 'x' | Mở file ở chế độ tạo mới, nếu file đã tồn tại thì sẽ bị báo lỗi |
| 'a' | Mở file ở chế độ ghi tiếp theo ở phân cuối của file, tạo mới nếu nó không tồn tại |
| 't' | Mở file ở chế độ dữ liệu kiểu text (mặc định) |
| 'b' | Mở file ở chế độ dữ liệu kiểu nhị phần binary |
| '+' | Mở file ở cả hai chế độ đọc và ghi |
| **Mode** | **Description** |

**Ví dụ**

|  |  |
| --- | --- |
| 1  2  3 | f = open("test.txt")      # equivalent to 'r' or 'rt'  f = open("test.txt",'w')  # write in text mode  f = open("img.bmp",'r+b') # read and write in binary mode |

Như vậy bạn có thể sử dụng một hoặc nhiều chế độ trong cùng một phiên mở file.

Không giống như các ngôn ngữ khác, ký tự 'a' không phải là số 97 trong bảng mã ASCII (hoặc mã hóa tương đương khác), nó sẽ phụ thuộc vào nền tảng mà bạn đang sư dụng. Trong Window thì là 'cp1252' nhưng trong Linux thì 'utf-8', vì vậy, chúng ta không nên sử dụng mã hóa mặc định mà chỉ rõ trong lúc mở file.

**Ví dụ**

|  |  |
| --- | --- |
| 1 | f = open("test.txt",mode = 'r',encoding = 'utf-8') |

### Đóng kế nối với close() trong Python

Khi ta mở một file và xử lý xong thì nên đóng file đó lại để đảm bảo tính an toàn, và để làm điều này thì ta sử dụng hàm close().

**Ví dụ**

|  |  |
| --- | --- |
| 1  2  3 | f = open("test.txt",encoding = 'utf-8')  # perform file operations  f.close( |

Phương thức này sẽ không an toàn trong một số trường hợp, nó sẽ không tự đóng file nếu trong quá trình mở và ghi bị lỗi. Để khắc phục vấn đề này thì ta sử dụng từ khóa with, nó sẽ đảm bảo răng file sẽ luôn được đóng khi lệnh kết thúc.

**Ví dụ**

|  |  |
| --- | --- |
| 1  2 | with open("test.txt",encoding = 'utf-8') as f:     # perform file operations |

Và ta cũng không cần phải gọi hàm close() vì nó tự động thực hiện ngầm.

### Ghi file trong Python

Để ghi vào một file bằng Python thì bắt buộc chúng ta cần mở nó ở chế độ ghi 'w', nối thêm 'a' hoặc tạo mới 'x'. Chúng ta cần cẩn thận với chế độ 'w' vì nó sẽ ghi đè lên tệp nếu nó đã tồn tại, tất cả dữ liệu trước đó bị xóa.

**Ví dụ**

|  |  |
| --- | --- |
| 1  2  3  4 | with open("test.txt",'w',encoding = 'utf-8') as f:     f.write("my first file\n")     f.write("This file\n\n")     f.write("contains three lines\n") |

Chương trình này sẽ tạo một file tên là test.txt nếu nó chưa tồn tại, còn nếu đã tồn tại thì dữ liệu trước đây sẽ bị xóa hết.

### Đọc dữ liệu của file trong Python

Để đọc dữ liệu của file thì chúng ta phải mở nó ở chế độ reading (r), sau đó có thể sử dụng hàm read(size) để để đọc dữ liệu của file, nếu bạn không truyền size thì nó sẽ đọc từ đầu đến cuối file.

**Ví dụ**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | >>> f = open("test.txt",'r',encoding = 'utf-8')  >>> f.read(4)    # read the first 4 data  'This'    >>> f.read(4)    # read the next 4 data  ' is '    >>> f.read()     # read in the rest till end of file  'my first file\nThis file\ncontains three lines\n'    >>> f.read()  # further reading returns empty sting  '' |

Làm thế nào để Python biết được là xuống dòng? Nó sẽ nhận biết thông qua ký tự thoát \n.

Để lấy số dòng hiện tại đang đọc thì ta dùng hàm tell(), và dùng hàm seek() để di chuyển con trỏ đọc tới một vị trí nào đó.

**Ví dụ**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | >>> f.tell()    # get the current file position  56    >>> f.seek(0)   # bring file cursor to initial position  0    >>> print(f.read())  # read the entire file  This is my first file  This file  contains three lines |

Ngoài ra để đọc từng dòng thì ta có thể kết hơp với vòng lặp như ví dụ dưới đây.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | >>> for line in f:  ...     print(line, end = '')  ...  This is my first file  This file  contains three lines |

Mỗi lần lặp chính là một dòng dữ liệu của file.

Để đọc một dòng duy nhất thì ta dùng hàm readline(), hàm này sẽ đọc dòng mới và sẽ ngưng khi kết thúc dòng đó.

**Ví dụ**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | >>> f.readline()  'This is my first file\n'    >>> f.readline()  'This file\n'    >>> f.readline()  'contains three lines\n'    >>> f.readline()  '' |

Để đọc tất cả các dòng và trả về dạng list thì ta dùng hàm readlines().

|  |  |
| --- | --- |
| 1  2 | >>> f.readlines()  ['This is my first file\n', 'This file\n', 'contains three lines\n'] |

### Lời kết

Trên là những thao tác cơ bản xử lý file trong Python, những thao tác này rất đơn giản nhưng lại rất hữu ích, hy vọng bài viết sẽ giúp được bạn trong quá trình học Python.

## Sơ lược Date/Time trong Python

Trong bài này chúng ta sẽ tìm hiểu cách xử lý thời gian (date - time) trong Python như: Cách import object date, lấy thời gian hiện tai, định dạng thời gian, ...

Trong các ứng dụng thực tế, có những lúc chúng ta cần phải làm việc với ngày tháng và thời gian. Ví dụ cần viết một ứng dụng đặt lịch hẹn bằng ngôn ngữ Python thì ta phải biết cách lấy thời gian trên hệ thống để xử lý.

Trong python, date không phải là kiểu dữ liệu, nhưng chúng ta có thể làm việc với các đối tượng ngày bằng cách import mô-đun có tên là datetime, time và calendar.

**Mục lục**

* [1. Mốc thời gian nhỏ nhất trong Python](https://freetuts.net/so-luoc-datetime-trong-python-2061.html#goto-h2-0)
* [2. Cách lấy thời gian hiện tại - localtime()](https://freetuts.net/so-luoc-datetime-trong-python-2061.html#goto-h2-1)
* [3. Time tuple là gì?](https://freetuts.net/so-luoc-datetime-trong-python-2061.html#goto-h2-2)
* [4. Hàm asctime() thay đổi format time](https://freetuts.net/so-luoc-datetime-trong-python-2061.html#goto-h2-3)
* [5. Hàm sleep() trong Python](https://freetuts.net/so-luoc-datetime-trong-python-2061.html#goto-h2-4)
* [6. Module datetime trong Python](https://freetuts.net/so-luoc-datetime-trong-python-2061.html#goto-h2-5)
  + [1. Tạo date object](https://freetuts.net/so-luoc-datetime-trong-python-2061.html#goto-h3-0)
  + [2. So sánh 2 ngày khác nhau](https://freetuts.net/so-luoc-datetime-trong-python-2061.html#goto-h3-1)
* [7. Calendar module trong Python](https://freetuts.net/so-luoc-datetime-trong-python-2061.html#goto-h2-6)

### Mốc thời gian nhỏ nhất trong Python

Cũng như hầu hết các ngôn ngữ lập trình khác, Python lấy ngày**1/1/1970** làm mốc nhỏ nhất. Hàm time() sẽ trả về tổng số mili giây đã trôi qua tính từ ngày 1/1/1970.

Xem ví dụ dưới đây để hiểu rõ hơn:

|  |  |
| --- | --- |
| 1  2  3  4  5 | import time;    #In ra tổng số mili giây kể từ ngày 1/1/1970    print(time.time()) |

Kết quả trả về tính đến thời điểm hiện tại:

|  |
| --- |
| 1545124470.9151752 |

### Cách lấy thời gian hiện tại - localtime()

Để lấy thời gian hiện tại trên hệ thống thì ta sử dụng hàm localtime(). Cách sử dụng rất đơn giản, bạn chỉ cần import module time và gọi hàm này một cách bình thường.

Xem ví dụ sau:

|  |  |
| --- | --- |
| 1  2  3  4  5 | import time;    #returns a time tuple    print(time.localtime(time.time())) |

Kết quả nó trả về là một tuple.

|  |
| --- |
| time.struct\_time(tm\_year=2018, tm\_mon=12, tm\_mday=18, tm\_hour=15, tm\_min=1,  tm\_sec=32, tm\_wday=1, tm\_yday=352, tm\_isdst=0) |

### Time tuple là gì?

Time tuple là bộ các đơn vị thời gian, gồm 9 thông số như sau:

|  |  |  |
| --- | --- | --- |
| **Index** | **Attribute** | **Values** |
| 0 | Year | 4 số nguyên(for example 2018) |
| 1 | Month | 1 đến 12 |
| 2 | Day | 1 đến 31 |
| 3 | Hour | 0 đến 23 |
| 4 | Minute | 0 đến 59 |
| 5 | Second | 0 đến 60 |
| 6 | Day of weak | 0 đến 6 |
| 7 | Day of year | 1 đến 366 |
| 8 | Daylight savings | -1, 0, 1 , or -1 |

### Hàm asctime() thay đổi format time

Bạn có thể thay đổi định dạng thời gian bằng cách sử dụng hàm asctime() năm trong **module time**. Nó sẽ trả về thời gian được định dạng của đối tượng time đang sử dụng hàm này.

|  |  |
| --- | --- |
| 1  2  3  4  5 | import time;    #Trả về định dạng time    print(time.asctime(time.localtime(time.time()))) |

Kết quả:

|  |
| --- |
| Tue Dec 18 15:31:39 2019 |

### Hàm sleep() trong Python

Hàm sleep() trong Python được dùng để dừng thực thi chương trình trong một khoảng thời gian nhất định. Số thời gian bị hoãn phụ thuộc vào giá trị mà bạn truyền vào hàm này.

Đơn vị của tham số thời gian đầu vào tính bằng giây, có thể số nguyên hoặt số thực.

**Ví dụ**: Tạo vòng lặp, mỗi lần lặp cho nghỉ 1 giây.

|  |  |
| --- | --- |
| 1  2  3  4  5 | import time  for i in range(0,5):      print(i)      #Mỗi lần lặp sẽ nghỉ 1 giây, sau đó in kết quả ra màn hình      time.sleep(1) |

### Module datetime trong Python

**Module datetime** cho phép ta tạo đối tượng thời gian và có thể tùy chỉnh theo ý mình. Đối tượng này được sử dụng rất nhiều trong thực tế. Ví dụ bạn làm lịch hẹn giờ thì chắc chắn phải dùng module date để lấy thời gian hiện tai, datetime để lấy thời gian hẹn, sau đó so sánh và xử lý hiển thị kết quả.

Tương tự, ta phải import datetime thì mới sử dụng các phương thức - hàm ở trong module này được.

**Ví dụ**: In ra thời gian hiện tại

|  |  |
| --- | --- |
| 1  2  3  4  5 | import datetime;    #Trả về object time hiện tại    print(datetime.datetime.now()) |

**Kết quả**:

|  |
| --- |
| 2018-12-18 16:16:45.462778 |

#### Tạo date object

Ta có thể tạo một date object bằng cách truyền giá trị ngày vào ba tham số của hàm datetime.

Như ví dụ dưới đây mình truyền vào 10/12/2018.

|  |  |
| --- | --- |
| 1  2  3 | import datetime;    print(datetime.datetime(2018,12,10)) |

Hoặc truyền nhiều hơn 3 tham số:

|  |  |
| --- | --- |
| 1 | print(datetime.datetime(2018,12,10,14,15,10)) |

#### 2. So sánh 2 ngày khác nhau

Để so sánh hai ngày hơn kém nhau thì ta có thể sử dụng các toán tử so sánh (>, >=, <, <=, =).

**Xem ví dụ dưới đây để hiểu rõ hơn**.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | from datetime import datetime as dt  #So sánh thời gian. Nếu thời gian nằm trong khoảng 8AM và 4PM, thì in ra màn hình thời gian việc, ngược lai là thời gian nghỉ ngơi  if dt(dt.now().year,dt.now().month,dt.now().day,8)<dt.now()<dt(dt.now().year,dt.now().month,dt.now().day,16):      print("Thời gian lam việc ....")  else:      print("Thời gian nghỉ ngơi") |

### Calendar module trong Python

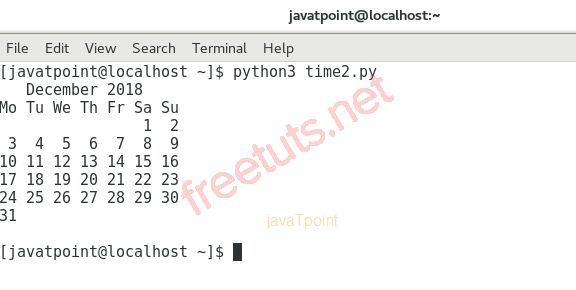
Python cung cấp một module về xử lý lịch (calendar) chứa các phương thức khác nhau để làm việc với lịch.

Hãy xem xét ví dụ sau để in lịch của tháng cuối năm 2018.

**Ví dụ**

|  |  |
| --- | --- |
| 1  2  3  4 | import calendar;  cal = calendar.month(2018,12)  #In lịch năm 2018  print(cal) |

Kết quả:



Nếu bạn muốn in ra lịch của cả năm thì làm như sau:

|  |  |
| --- | --- |
| 1  2  3  4 | import calendar    #In lịch năm 2019  calendar.prcal(2019) |

