LẬP TRÌNH PYTHON TỪ CƠ BẢN ĐẾN NÂNG CAO

Thực hiện: TS.Trần Bình Long

NỘI DUNG

1. Đối tượng Iterator
2. Generator
3. Closure
4. Decorator
5. @property
6. RegEx

Iterator

Lấy từng phần tử, khi sử dụng vòng lặp để có một nhóm phần tử.

Iterator có hai phương thức là ***__iter__()*** và ***__next__()***, gọi là giao thức iterator (Iterator Protocol).

- Phương thức ***__iter__*** trả về đối tượng iterator, dùng cho đối tượng "**iterable**" và **iterator** sử dụng các câu lệnh for và in.

- Phương thức ***__next__*** trả về phần tử tiếp theo. Nếu không còn phần tử nào sẽ báo lỗi StopIteration.

Iterator

Iterable object khi sử dụng sẽ trả về một **iterator**, ví dụ: Chuỗi, List, Tuple.

Iter() là hàm dựng sẵn trong Python nhận đầu vào là một đối tượng iterable và trả về kết quả là một iterator.

```
>>> # Khai bao list
    my_list = [4, 7, 0, 3]
    # lay mot iterator bang cach su dung iter()
    my_iter = iter(my_list)
    ## su dung next()
    print(next(my_iter))
    #Output 4
    print(next(my_iter))
    #Output 7
```

Iterator

```
>>> ## next(obj) chinh la obj.__next__()
    print(my_iter.__next__())
    #prints 0
    print(my_iter.__next__())
    #prints 3
    next(my_iter)
    ## Xay ra loi StopIteration vi het gia tri tra ve
    Traceback (most recent call last):
      File "<stdin>", line 24, in <module>
        next(my_iter)
    StopIteration
```

Iterator

Tương tự kết quả trả về dùng vòng lặp *for*

```
>>> for element in my_list:
        print(element)
```

Kết quả:

```
>>> 4
    7
    0
    3
```

Cách vòng lặp hoạt động

Vòng lặp *for* lặp lại tự động thông qua việc sử dụng danh sách.
Vòng lặp *for* lặp lại trên bất kỳ iterable nào, xem kỹ hơn về cách vòng lặp for được thực hiện trong Python

```
>>> for element in iterable:
        # do something with element
```

Được thực hiện tương tự như:

Cách vòng lặp hoạt động

```
>>> # iter_obj là một iterator object tạo từ iterable
    iter_obj = iter(iterable)
    # vòng lặp
    while True:
        try:
            # sử dụng next
            element = next(iter_obj)
        except StopIteration:
            # nếu xảy ra lỗi StopIteration thì vòng lặp sẽ
            break
```

Cách vòng lặp hoạt động

Ví dụ trên, trong vòng lặp ***for*** tạo một **iterator object** tên là ***iter_obj*** bằng cách gọi ***iter()*** trên **iterable**.

Như vậy, vòng lặp ***for*** ở đây chính là vòng lặp ***while*** vô tận. ***Next()*** bên trong vòng lặp lấy ra các phần tử để thực hiện các câu lệnh ở ***for*** loop.

Khi lấy hết các giá trị thì ngoại lệ **StopIteration** sẽ được sinh ra và vòng lặp kết thúc.

Xây dựng trình vòng lặp Iterator

Xây dựng iterator là một class, chỉ cần thực hiện các phương thức ***__iter__()*** và ***__next__()***.

```
>>> class PowTwo:
        def __init__(self, max = 0):
            self.max = max
        def __iter__(self):
            self.n = 0
            return self
        def __next__(self):
            if self.n <= self.max:
```

Xây dựng trình vòng lặp Iterator

```
>>>             result = 2 ** self.n
                self.n += 1
                return result
            else:
                raise StopIteration
```

Phương thức ***__iter__*** sẽ làm đối tượng trở thành đối tượng iterable. Giá trị trả về của ***__iter__*** là một iterator.

Phương thức ***__next__*** và trả về ***StopIteration*** nếu không còn phần thử nào nữa.

Tạo ra một iterator như sau:

Xây dựng trình vòng lặp Iterator

```
>>> a = PowTwo(4)
>>> i = iter(a)
>>> print(next(i))
    1
>>> print(next(i))
    2
>>> print(next(i))
    4
>>> print(next(i))
    8
>>> print(next(i))
    16
>>> print(next(i))
     Traceback (most recent call last):
    ...
    StopIteration
```

Xây dựng trình vòng lặp Iterator

Sử dụng vòng lặp *for* để lặp iterator

```
>>> for i in PowTwo(5):
        print(i)
```

Kết quả:

```
>>> 1
    2
    4
    8
    16
    32
```

Iterator lặp vô tận

Không phải tất cả các đối tượng **iterator** đều sẽ được gọi hết các phần tử và kết thúc khi không còn phần tử. Có một số trường hợp **iterator** sẽ lặp vô tận. Ví dụ:

```
>>> int()
    0
>>> inf = iter(int,1)
>>> next(inf)
    0
>>> next(inf)
    0
```

Iterator lặp vô tận

hàm *int()* luôn trả về 0. Vì vậy, việc truyền hàm dưới dạng ***iter(int, 1)*** sẽ trả về một **iterator** lặp cho đến khi giá trị trả về bằng 1. Điều này không bao giờ xảy ra và đây chính là iterator lặp vô tận. Cần lưu ý khi xử lý trong các trường hợp này.

Ngoài ra, có thể tự xây dựng iterator lặp vô tận. Ví dụ sau đây lặp vô tận và trả về các số lẻ vì không có điều kiện dừng.

Iterator lặp vô tận

```
>>> class InfIter:
        def __iter__(self):
            self.num = 1
            return self
        def __next__(self):
            num = self.num
            self.num += 2
            return num
```

Chạy chương trình:

```
>>> a = iter(InfIter())
>>> print(next(a))
    1
>>> print(next(a))
```

Iterator lặp vô tận

```
      3
>>> print(next(a))
      5
>>> print(next(a))
      7
```

Ưu điểm của việc sử dụng Iterator lặp là tiết kiệm tài nguyên. Như ví dụ trên, có thể nhận được tất cả các số lẻ mà không lưu trữ toàn bộ hệ thống số ở bộ nhớ.

Generator

Xây dựng iterator, phải triển khai class với phương thức ***__iter__()*** và ***__next__()***, theo dõi các tình trạng, ***StopIteration*** khi không có giá trị nào trả về...

Generator là cách đơn giản để tạo iterator. **Generator là một hàm trả về một đối tượng (iterator) có thể lặp lại (một giá trị tại một thời điểm).**

Tạo ra đối tượng kiểu danh sách, chỉ duyệt qua các phần tử của generator một lần duy nhất, **generator** không lưu dữ liệu trong bộ nhớ, mỗi lần lặp sẽ tạo phần tử tiếp theo trong dãy và trả về phần tử đó.

Tạo Generator

Tạo generator dùng từ khóa def giống định nghĩa hàm. Trong generator, câu lệnh **yield** để trả về các phần tử thay **return**.

Nếu một hàm chứa ít nhất một **yield** (có thể có nhiều **yield** và cả **return**) thì chắc chắn đây là hàm **generator**. Trong trường hợp này, cả **yield** và **return** sẽ trả về các giá trị từ hàm.

return sẽ chấm dứt hàm, còn **yield** chỉ tạm dừng các trạng thái bên trong hàm và sau đó sẽ tiếp tục khi được gọi.

Tạo Generator

Khi gọi phương thức ***__next__()*** lần thứ nhất, **generator** thực hiện việc tính giá trị, khi gặp từ khóa **yield** sẽ trả về các phần tử tại ví trí đó, khi gọi phương thức ***__next__()*** lần thứ hai **generator** sẽ bắt đầu ngay phía sau từ khóa **yield** thứ nhất. Cứ như thế **generator** tạo ra các phần tử trong dãy, cho đến khi không còn từ khóa **yield** nào nữa thì giải phóng **exception *StopIteration***.

So sánh hàm generator và hàm thông thường

Khác nhau giữa hàm generator và hàm thường:

- Hàm generator chứa một hoặc nhiều câu lệnh yield.
- Khi được gọi, generator trả về một đối tượng *(iterator)* nhưng không thực thi ngay lập tức.
- Các phương thức ***__iter__()*** và ***__next__()*** được triển khai tự động. Vì vậy, có thể lặp qua các mục bằng cách sử dụng *next().*
- **Yield** sẽ tạm dừng hàm, các biến cục bộ và trạng thái được ghi nhớ giữa các lệnh gọi liên tiếp. Mỗi lần lệnh yield chạy, sẽ sinh ra một giá trị mới.
- Cuối cùng, khi hàm kết thúc, ***StopIteration*** sẽ xảy ra nếu tiếp tục gọi hàm.

Hàm generator

Hàm generator my_gen() với câu lệnh yield .

```
>>> # Hàm generator đơn giản
    def my_gen():
        n = 1
        print('Doan text nay duoc in dau tien')
        # Hàm Generator chứa các câu lệnh yield
        yield n
        n += 1
        print('Doan text nay duoc in lần hai')
        yield n
        n += 1
        print('Doan text nay duoc in cuoi cung')
        yield n
```

Hàm generator

Output:

```
>>> # Trả về một đối tượng nhưng không bắt đầu thực thi ngay lập tức
>>> a = my_gen()
>>> # Lặp qua các mục bằng cách sử dụng next()
>>> next(a)
    Doan text nay duoc in dau tien
    1
>>> # Yield sẽ tạm dừng hàm, quyền điều khiển chuyển đến người gọi các biến cục bộ và trạng thái của chúng được ghi nhớ giữa các lệnh gọi liên tiếp.
>>> next(a)
    Doan text nay duoc in thu hai
    2
```

Hàm generator

```
>>> next(a)
    Doan text nay duoc in cuoi cung
    3
>>> # Cuối cùng, khi hàm kết thúc, StopIteration sẽ xảy ra nếu tiếp tục gọi hàm.
>>> next(a)
    Traceback (most recent call last):
     ...
     StopIteration
```

Hàm generator

Ví dụ trên, giá trị của biến n được ghi nhớ giữa các lần gọi, không kết thúc như các hàm thường ngay sau mỗi lần gọi.

Khi gọi phương thức ***next()*** lần thứ nhất, **generator** thực hiện tính giá trị rồi trả về phần tử tại ví trí đó, khi gọi phương thức ***next()*** lần thứ hai, **generator** bắt đầu ngay phía sau từ khóa yield thứ nhất.

Generator tạo ra các phần tử trong dãy, cho đến khi không còn gặp từ khóa yield nào nữa thì giải phóng exception *StopIteration*

Hàm generator

Tạo một đối tượng generator khác bằng cách sử dụng đối tượng như a = my_gen().

Lưu ý: Có thể sử dụng generator trực tiếp cho các vòng lặp for.

Vòng lặp *for* lấy một iterator và lặp lại nó bằng hàm *next(),* tự động kết thúc khi *StopIteration* xảy ra.

Hàm generator

```
>>> # Hàm generator đơn giản
    def my_gen():
        n = 1
        print('Doan text nay duoc in dau tien')
        # Hàm Generator chứa câu lệnh yield
        yield n
        n += 1
        print('Doan text nay duoc in thu hai')
        yield n
        n += 1
        print('Doan text nay duoc in cuoi cung')
        yield n
```

Hàm generator

```
>>> # Sử dụng vòng lặp for
    for item in my_gen():
        print(item)
```

Chạy chương trình, kết quả:

```
>>> Doan text nay duoc in dau tien
    1
    Doan text nay duoc in thu hai
    2
    Doan text nay duoc in cuoi cung
    3
```

Generator với vòng lặp

generator đảo ngược chuỗi

```
>>> def rev_str(my_str):
        length = len(my_str)
        for i in range(length - 1,-1,-1):
            yield my_str[i]
# Vòng lặp for đảo ngược chuỗi
>>> for char in rev_str("hello"):
        print(char)
# Output:# o
          # l
          # l
          # e
          # h
```

Dùng hàm *range()* để lấy chỉ mục theo thứ tự ngược lại trong vòng lặp *for.*

Biểu thức generator

Generator được tạo dễ dàng khi sử dụng **biểu thức generator**.

Giống **Lambda** tạo hàm vô danh, **generator** cũng tạo biểu thức **generator** vô danh.

Cú pháp tương tự như cú pháp **list comprehension**, nhưng dấu ngoặc vuông được thay thế bằng dấu ngoặc tròn.

List comprehension trả về một list, biểu thức **generator** trả về một generator tại thời điểm khi được yêu cầu. Biểu thức **generator** sử dụng ít bộ nhớ hơn, hiệu quả hơn so với list comprehension tương đương

Biểu thức generator

```
>>># Khởi tạo danh sách
      my_list = [1, 3, 6, 10]
      # bình phương mỗi phần tử bằng cách sử dụng
list comprehension
      [x**2 for x in my_list]
      # Output: [1, 9, 36, 100]
      (x**2 for x in my_list)    # dùng biểu thức generator
      # Output: <generator object <genexpr> at
0x0000000002EBDAF8>
```

Biểu thức generator không tạo ra kết quả mà trả về đối tượng generator, mỗi lần lặp sẽ tạo phần tử tiếp theo trong dãy và trả về phần tử đó.

Biểu thức generator

```
>>> # Khởi tạo danh sách
      my_list = [1, 3, 6, 10]
      a = (x**2 for x in my_list)
      print(next(a))
      # Output: 1
      print(next(a))
      # Output: 9
      print(next(a))
      # Output: 36
      print(next(a))
      # Output: 100
      next(a)
      # Output: StopIteration
```

Biểu thức generator

Biểu thức generator bên trong hàm có thể bỏ qua các dấu ngoặc tròn

```
>>> sum(x**2 for x in my_list)
    146
>>> max(x**2 for x in my_list)
    100
```

Generator đơn giản hóa code

Generator giúp code rõ ràng và ngắn hơn so với class iterator tương tự.

```
>>> class PowTwo:
        def __init__(self, max = 0):
            self.max = max
        def __iter__(self):
            self.n = 0
            return self
        def __next__(self):
            if self.n > self.max:
                raise StopIteration
            result = 2 ** self.n
            self.n += 1
            return result
```

Generator đơn giản hóa code

Đoạn code này dài hơn so với dùng hàm generator.

```
>>> def PowTwoGen(max = 0):
        n = 0
        while n < max:
            yield 2 ** n
            n += 1
```

Generator thực hiện ngắn gọn hơn.

Generator sử dụng ít bộ nhớ

Hàm thường khi trả về list sẽ lưu toàn bộ list trong bộ nhớ, điều này gây hao tốn tài nguyên khi phải sử dụng dung lượng bộ nhớ lớn.

Generator sử dụng ít bộ nhớ hơn vì chỉ tạo kết quả khi được gọi tới, sinh ra một phần tử tại một thời điểm, đem lại hiệu quả nếu không duyệt quá nhiều lần.

Generator tạo các list vô tận

Generator là cách tốt để tạo ra luồng dữ liệu vô tận. Các luồng vô tận này không cần lưu trữ toàn bộ trong bộ nhớ vì generator chỉ tạo ra một phần tử tại một thời điểm, nên có thể biểu thị luồng dữ liệu vô tận.

```
>>> def all_even():
        n = 0
        while True:
            yield n
            n += 2
```

Việc lựa chọn sử dụng generator phụ thuộc vào thực tế yêu cầu của công việc.

Định nghĩa hàm Closure

Closure được tạo ra bởi hàm lồng nhau, ghi nhớ không gian nơi tạo ra

```
>>> def print_msg(msg):
    # Hàm bên ngoài
        def printer():
        # Hàm lồng nhau
            print(msg)
        return printer
    another = print_msg("Hello")
    another()
```

Hàm Closure

Khi *print_msg* gọi với chuỗi "Hello", hàm con printer sẽ được định nghĩa cùng với giá trị của *msg* cục bộ. Sau đó, hàm *another()* được gọi. Khi thực thi hàm, *printer* tham chiếu đến *msg ,* giá trị này sẽ được ghi nhớ ngay cả khi hàm *print_msg* đã kết thúc.

```
>>> del print_msg
>>> another()
    Hello
>>> print_msg("Hello")
    Traceback (most recent call last):
    ...
    NameError: name 'print_msg' is not defined
```

Hàm Closure

Lưu ý: closure được tạo bởi hàm lồng nhau nhưng không phải hàm lồng nhau nào cũng là closure.

Closure chỉ được tạo khi hàm con truy cập đến những biến cục bộ trong pham vi được đóng bởi hàm cha.

Điều kiện để có Closure

Closure sử dụng khi một hàm lồng nhau tham chiếu đến một giá trị trong phạm vi (scope), nơi nó được định nghĩa.
Tiêu chí để tạo Closure:

- Có một hàm lồng nhau (hàm được định nghĩa bên trong một hàm khác).
- Hàm lồng nhau phải tham chiếu đến một giá trị được xác định trong hàm kèm theo.
- Hàm kèm theo phải trả kết quả về hàm lồng nhau.

Khi nào nên sử dụng Closure

Closure có thể sử dụng các giá trị global và cung cấp một số dạng dữ liệu ẩn. Closure cũng có thể cung cấp các giải pháp, tạo ra các hàm có nhiều tính chất của lập trình hướng đối tượng để giải quyết vấn đề.

Khi định nghĩa class chỉ với vài phương thức, closure được sử dụng như một giải pháp thay thế nhẹ nhàng hơn lập trình hướng đối tượng. Tuy nhiên khi các thuộc tính và phương thức tăng lên, sử dụng lập trình hướng đối tượng sẽ là giải pháp tốt hơn.

Khi nào nên sử dụng Closure

Sử dụng closure thích hợp hơn lập trình hướng đối tượng với việc xác định lớp và tạo các đối tượng.

```
>>> def make_multiplier_of(n):
        def multiplier(x):
            return x * n
        return multiplier
>>> # He so 3
    times3 = make_multiplier_of(3)
    # He so 5
    times5 = make_multiplier_of(5)
    # Output: 27
    print(times3(9))
```

Khi nào nên sử dụng Closure

```
>>> # Output: 15
>>> print(times5(3))
    # Output: 30
    print(times5(times3(2)))
```

Trong những ứng dụng của hàm lồng nhau, closure chính là ứng dụng quan trọng nhất. Nếu biết sử dụng đúng cách, hiệu suất sẽ cao và dễ bảo trì.

Decorator

Decorator là hàm **nhận tham số đầu vào là một hàm khác và mở rộng tính năng cho hàm đó mà không thay đổi nội dung của nó.**

Còn được gọi là metaprogramming - siêu lập trình, "Code sinh ra code", nghĩa là khi viết chương trình và chương trình này sẽ sinh ra, điều khiển các chương trình khác hoặc làm một phần công việc tại thời điểm biên dịch.

Điều kiện để có Decorator

Hàm là khái niệm cơ bản trong lập trình, **hàm cũng là đối tượng.** Tên hàm là định danh ràng buộc với đối tượng. Một đối tượng hàm cũng có thể được liên kết với nhiều tên khác nhau.

```
>>> def first(msg):
        print(msg)
    first("Hello")
    second = first
    second("Hello")
```

Hàm first và second đều trả về cùng một output. Ở đây, first và second đề cập đến cùng một đối tượng hàm.

Decorator

Hàm có thể được truyền dưới dạng tham số cho một hàm khác (tương tự ***map***, ***filter*** và ***reduce***).

Hàm lấy hàm khác làm tham số đầu vào được gọi là **hàm bậc cao** (higher-order functions). Ví dụ:

```
>>> def inc(x):
        return x + 1
    def dec(x):
        return x - 1
    def operate(func, x):
        result = func(x)
        return result
```

Decorator

Gọi hàm:

```
>>> print(operate(inc,3))
    4
>>> print(operate(dec,3))
    2
```

Hơn nữa, hàm có thể trả về kết quả của một hàm khác

```
>>> def is_called():
        def is_returned():
            print("Hello")
        return is_returned
     new = is_called()
>>> #Outputs "Hello"
    new()
```

Decorator

is_returned() là một hàm lồng nhau, hàm này sẽ truy cập và trả về kết quả mỗi khi gọi hàm ***is_called()***.

Decorator là hàm nhận các hàm khác làm tham số, cho phép chạy một số đoạn code trước hoặc sau hàm chính mà kết quả không thay đổi.

```
>>> def make_pretty(func):
        def inner():
            print("I got decorated")
            func()
        return inner
    def ordinary():
        print("I am ordinary")
```

Decorator

Chạy codel:

```
>>> ordinary()
    I am ordinary
>>> # Thử hàm decorate trong hàm ordinary
>>> pretty = make_pretty(ordinary)
>>> pretty()
    I got decorated
    I am ordinary
```

Decorator

make_pretty() là **decorator**. Khi gọi pretty = make_pretty(ordinary) thì hàm ***ordinary()*** được **decorator** truyền vào làm tham số và hàm trả về tên là ***pretty()***.

Decorator đã thêm một hàm mới cho hàm ban đầu. Giống như gói quà, **decorator** là lớp bọc ở ngoài, bản chất của đối tượng được **decorator** truyền vào làm tham số (món quà bên trong) không thay đổi.

Decorator

Decorator là một hàm và gán chính nó: ordinary = make_pretty(ordinary), đây là cấu trúc phổ biến. Python có cú pháp đơn giản hơn:
Dùng ký hiệu @ cùng với tên hàm **decorator** và đặt trước định nghĩa của hàm được decorator. Ví dụ:

```
>>> @make_pretty
    def ordinary():
        print("I am ordinary")
```

tương đương với:

```
>>> def ordinary():
        print("I am ordinary")
    ordinary = make_pretty(ordinary)
```

Đây là cú pháp đặc biệt để thực hiện decorator

Tham số hàm decorator

Decorator trên đơn giản và hoạt động với các hàm không có tham số.

Decorator và hàm có tham số:

```
>>> def divide(a, b):
        return a/b
```

Hàm này có hai tham số, a và b, sẽ báo lỗi nếu b=0

```
>>> print(divide(2,5))
    0.4
>>> print(divide(2,0))
    Traceback (most recent call last):
    ...
    ZeroDivisionError: division by zero
```

Tham số hàm decorator

Khi gọi hàm được trả về bởi decorator, nếu truyền tham số cho hàm thì tham số này sẽ truyền cho hàm được decorate.

Tạo decorator để kiểm tra trường hợp có xảy ra lỗi hay không.

```
>>> def smart_divide(func):
        def inner(a,b):
            print("I am going to divide",a,"and",b)
            if b == 0:
                print("Whoops! cannot divide")
                return
            return func(a,b)
        return inner
```

Tham số hàm decorator

```
>>> @smart_divide
    def divide(a,b):
        return a/b
```

Chương trình sẽ trả về None nếu có lỗi phát sinh.

```
>>> print(divide(2,5))
    I am going to divide 2 and 5
    0.4
>>> print(divide(2,0))
    I am going to divide 2 and 0
    Whoops! cannot divide
```

Đây là cách sử dụng hàm decorator có tham số.

Tham số hàm decorator

Có thể xây dựng hàm **decorator** với số lượng tham số tùy ý, sử dụng cú pháp ***args** và ****kwargs**.

```
>>> def works_for_all(func):
        def inner(*args, **kwargs):
            print("I can decorate any function")
            return func(*args, **kwargs)
        return inner
```

Chuỗi Decorator

Decorator có thể tạo thành chuỗi **decorator,** một hàm có thể được decorate nhiều lần với các decorator giống hoặc khác nhau, chỉ cần đặt decorator trước hàm.

```
>>> def star(func):
        def inner(*args, **kwargs):
            print("*" * 30)
            func(*args, **kwargs)
            print("*" * 30)
        return inner
    def percent(func):
        def inner(*args, **kwargs):
            print("%" * 30)
```

Chuỗi Decorator

```
>>>         func(*args, **kwargs)
            print("%" * 30)
        return inner
    @star
    @percent
    def printer(msg):
        print(msg)
    printer("Hello")
# Output:
******************************
%%%%%%%%%%%%%%%%
Hello
%%%%%%%%%%%%%%%%
******************************
```

Chuỗi Decorator

Cú pháp:

```
>>> @star
      @percent
      def printer(msg):
          print(msg)
```

tương đương với:

```
>>> def printer(msg):
          print(msg)
      printer = star(percent(printer))
```

Chuỗi Decorator

Thứ tự của decorator cần quan tâm, nếu đảo ngược:

```
>>> @percent
      @star
      def printer(msg):
          print(msg)
     printer('Hello')
```

Kết quả sẽ khác:

```
%%%%%%%%%%%%%%
******************************
Hello
******************************
%%%%%%%%%%%%%%
```

@property decorator

Decorate một hàm bằng decorator, một số decorator được tích hợp sẵn: ***@property***.

Mục đích của các decorator là thay đổi các phương thức, thuộc tính class mà không cần thực hiện thay đổi code.

Sử dụng @property decorator

```
>>> class Student:
        def __init__(self, name, marks):
            self.name = name
            self.marks = marks
            self.gotmarks = self.name + ' obtained ' +
self.marks + 'marks'
    st = Student("Jaki", "25")
    print(st.name)
    print(st.marks)
    print(st.gotmarks)
# Output: Jaki
            25
            Jaki obtained 25 marks
```

Sử dụng @property decorator

Bây giờ, thay đổi thuộc tính name của class *student* , thêm 3 dòng sau vào code:

```
>>> st.name = "Anusha"
    print(st.name)
    print(st.gotmarks)
```

Chương trình sẽ trả về:

```
>>> Jaki
    25
    Jaki obtained 25 marks
     Anusha
    Jaki obtained 25 marks
```

Sử dụng @property decorator

Thuộc tính name đã thay đổi nhưng câu được tạo bởi thuộc tính gotmarks vẫn giống như lúc đặt trong quá trình khởi tạo đối tượng *student.*

Bây giờ muốn gotmarks thay đổi khi tên sinh viên được cập nhật. Sử dụng **@property decorator**:

```
>>> class Student:
        def __init__(self, name, marks):
            self.name = name
            self.marks = marks
# self.gotmarks = self.name + ' obtained ' + self.marks
+ ‘marks’
        def gotmarks(self):
            return self.name + ' obtained ' + self.marks +
‘marks’
```

Sử dụng @property decorator

```
>>> st = Student("Jaki", "25")
      print(st.name)
      print(st.marks)
      print(st.gotmarks())
      st.name = "Anusha"
      print(st.name)
      print(st.gotmarks())
# Output:
>>> Jaki
      25
      Jaki obtained 25 marks
       Anusha
       Anusha obtained 25 marks
```

Sử dụng @property decorator

Yêu cầu đã được giải quyết. Đã loại bỏ thuộc tính gotmarks khỏi hàm và thêm phương thức gotmarks().

Với thay đổi này, người sử dụng class sẽ gặp rắc rối vì cần phải thay thế tất cả các thuộc tính gotmarks bằng hàm gotmarks(). Nếu code quá dài việc sửa đổi sẽ phức tạp, mất công và dễ nhầm lẫn.

Sử dụng @property decorator

Sử dụng @property decorator.

```
>>> @property
      Def gotmarks(self):
         return self.name + ‘obtained’ + self.marks + ‘marks’
```

Xóa () phía sau gotmarks khi in kết quả, Output như nhau.

Chỉ cần khai báo *@property* ở trên hàm gotmarks() để sẵn sàng sử dụng.

Với decorator này, tiếp tục giữ nguyên nội dung code cũ mà vẫn mở rộng tính năng cho hàm.

Thiết lập @property setter

Sử dụng @property (decorators) để khai báo setter. Muốn update thuộc tính *name* và *marks* khi thay đổi giá trị của *gotmarks.*

```
>>> class Student:
        def __init__(self, name, marks):
            self.name = name
            self.marks = marks
# self.gotmarks = self.name + ' obtained ' + self.marks + ‘marks’
        @property
        def gotmarks(self):
            return self.name + ' obtained ' + self.marks + ‘marks’
```

Thiết lập @property setter

```
>>> @gotmarks.setter
    def gotmarks(self, sentence):
        name, rand, marks = sentence.split(' ')
        self.name = name
        self.marks = marks
    st = Student("Jaki", "25")
    print(st.name)
    print(st.marks)
    print(st.gotmarks)
    print("####################")
    st.name = "Anusha"
    print(st.name)
    print(st.gotmarks)
```

Thiết lập @property setter

Để update giá trị của *name* và *marks* khi thay đổi giá trị *gotmarks,* dùng *setter* trong *@proprety decorator*. **@gotmarks.setter** áp dụng *setter* trên phương thức *gotmarks,* sau đó cập nhật giá trị của *name* và *marks.*

```
>>> Jaki
    25
    Jaki obtained 25 marks
    ####################
    Anusha
    Anusha obtained 25 marks
    ####################
   Golam obtained 36 marks
   Golam
   36
```

Regular Expression (RegEx)

Regular Expression (RegEx) còn gọi là **Biểu thức quy tắc** là đoạn các ký tự đặc biệt theo khuôn mẫu (pattern) nhất định, đại diện cho chuỗi hoặc tập các chuỗi. Ví dụ:

^a...s$

Xác định quy tắc RegEx: chuỗi bất kỳ có năm chữ cái, bắt đầu bằng a và kết thúc bằng s

Regular Expression

Biểu thức	Chuỗi ví dụ	Mô tả
^a...s$	abs	Không phù hợp vì chỉ có 3 ký tự
	alias	Phù hợp
	abyss	Phù hợp
	Alias	Không phù hợp vì chữ cái đầu viết hoa A
	An abacus	Không phù hợp vì chữ cái đầu viết hoa A và nhiều hơn 5 ký tự

Regular Expression

Regular Expression trong Python được thể hiện qua **module re**, muốn sử dụng Regular Expression phải import module re. Ví dụ:

```
>>> import re
    pattern = '^a...s$'
    test_string = 'abyss'
    result = re.match(pattern, test_string)
    if result:
        print("Tim kiem thanh cong.")
    else:
        print("Tim kiem khong thanh cong.")
```

Regular Expression

Sử dụng hàm **re.match()** để tìm *test_string* tương ứng với *pattern.* Phương thức trả về đối tượng tương ứng nếu tìm thấy, trả về None nếu không tìm thấy.

Nhiều ngôn ngữ hỗ trợ Regular Expression, như JavaScript, C#, Java, PHP, Ruby, SQL, Oracle, Perl...

Có nhiều hàm trong mudule re dùng để hoạt động với RegEx.

Trước khi đi sâu vào các hàm này, hãy tìm hiểu kĩ hơn về biểu thức quy tắc RegEx.

Cú pháp pattern sử dụng trong RegEx

Pattern là một đối tượng mẫu được biên dịch của biểu thức quy tắc.

Sử dụng các ký tự đặc biệt để chỉ định biểu thức quy tắc:

[ ] . ^ $ * + ? {} () \ |

là ký tự ^ và $.

Dấu ngoặc vuông [ ]

Dấu ngoặc vuông sử dụng để thể hiện tập các ký tự.

Cú pháp pattern

Biểu thức	**Chuỗi ví dụ**	**Mô tả**
[abc]	a	Khớp với ký tự a
	ac	Khớp với ký tự a hoặc c
	Hey Jude	Không khớp

[abc] sẽ khớp nếu chuỗi truyền có chứa bất kỳ ký tự a , b hoặc c.

Cú pháp pattern

Chỉ định phạm vi các ký tự bằng cách sử dụng dấu - bên trong dấu ngoặc vuông.

[a-e] tương tự với [abcde]
[1-4] tương tự với [1234]
[0-39] tương tự với [01239]

Nếu ký tự đầu tiên của tập hợp là ^ thì tất cả các ký tự không được định nghĩa trong tập hợp sẽ được so khớp.

[^abc] khớp với các chuỗi không có ký tự a , b hay c.

[^0-9] khớp với các chuỗi không có ký tự số.

Ký tự đặc biệt trong [ ] coi như ký tự thông thường.

[(+)] khớp với bất kỳ chuỗi nào có ký tự (, + hoặc)

Dấu chấm .

Dấu chấm khớp với bất kỳ ký tự đơn thông thường ngoại trừ ký tự tạo dòng mới '\n' .

Biểu thức	Chuỗi ví dụ	Mô tả
..	a	Không khớp vì chỉ có một ký tự
	ac	Khớp vì có hai ký tự
	acd	Khớp vì có hai ký tự trở lên

Dấu mũ ^

Biểu tượng dấu mũ ^ được sử dụng để khớp ký tự đứng đầu một chuỗi.

Biểu thức	Chuỗi ví dụ	Mô tả
^a	a	Khớp vì bắt đầu bằng a
	abc	Khớp vì bắt đầu bằng a
	bac	Không khớp vì a không nằm ở đầu tiên
^ab	abc	Khớp vì bắt đầu bằng ab
	acb	Không khớp, bắt đầu bằng a nhưng ký tự tiếp theo không phải b

Biểu tượng Dollar $

Biểu tượng Dollar $ được sử dụng để khớp ký tự kết thúc một chuỗi

Biểu thức	Chuỗi ví dụ	Mô tả
a$	a	Khớp vì kết thúc bằng a
	formula	Khớp vì kết thúc bằng a
	cab	Không khớp vì a không nằm ở vị trí cuối cùng

Dấu hoa thị *

Dấu hoa thị * khớp với chuỗi có hoặc không có ký tự được định nghĩa trước nó. Ký tự này có thể lặp lại nhiều lần.

Biểu thức	Chuỗi ví dụ	Mô tả
ma*n	mn	Khớp vì ký tự trước * có thể không xuất hiện
	man	Khớp vì có xuất hiện đầy đủ các ký tự
	maaaan	Khớp vì ký tự trước * có thể xuất hiện nhiều lần
	main	Không khớp vì không giống pattern, n không nằm kế a
	woman	Khớp vì có xuất hiện đầy đủ các ký tự

Dấu cộng +

Dấu cộng + khớp với chuỗi có một hoặc nhiều ký tự được định nghĩa trước nó. Ký tự này có thể lặp lại nhiều lần.

Biểu thức	Chuỗi ví dụ	Mô tả
ma+n	mn	Không khớp vì ký tự a trước + không xuất hiện
	man	Khớp vì có xuất hiện đầy đủ các ký tự
	maaaan	Khớp vì ký tự trước + có thể xuất hiện nhiều lần
	main	Không khớp vì không giống pattern, n không nằm kế a
	woman	Khớp vì có xuất hiện đầy đủ các ký tự

Dấu chấm hỏi ?

Dấu chấm hỏi khớp với chuỗi có hoặc không có ký tự được định nghĩa trước nó. Ký tự này **không thể** lặp lại nhiều lần, giới hạn số lượng với **một lần xuất hiện**.

Biểu thức	Chuỗi ví dụ	Mô tả
ma? n	mn	Khớp vì ký tự trước ? có thể không xuất hiện
	man	Khớp vì có xuất hiện đầy đủ các ký tự
	maaaan	Không khớp vì ký tự trước ? chỉ có thể xuất hiện 1 lần
	main	Không khớp vì không giống pattern, n không nằm kế a
	woman	Khớp vì có xuất hiện đầy đủ các ký tự

Dấu ngoặc nhọn {}

Dấu ngoặc nhọn sử dụng theo công thức: **{n,m}** , đại diện cho ký tự đằng trước nó có thể xuất hiện tối thiểu n lần vào tối đa m lần. n và m là số nguyên dương và n <= m.

Nếu bỏ trống n , giá trị này mặc định bằng 0.
Nếu bỏ trống m , giá trị này mặc định là vô hạn.

Dấu ngoặc nhọn {}

Biểu thức	Chuỗi ví dụ	Mô tả
a{2,3}	abc dat	Không khớp vì không thỏa mãn điều kiện
	abc daat	Khớp vì có xuất hiện 2 ký tự a (daat)
	aabc daaat	Khớp vì có xuất hiện 2 và 3 ký tự a (aabc và daaat)
	aabc daaaat	Khớp vì có xuất hiện 2 và 3 ký tự a (aabc và daaaat)

Dấu ngoặc nhọn {}

ví dụ: RegEx [0-9] {2, 4} khớp với chuỗi có tối thiểu 2 chữ số và tối đa không quá 4 chữ số.

Biểu thức	Chuỗi ví dụ	Mô tả
[0-9] {2,4}	ab123csde	Khớp vì thỏa mãn điều kiện: ab123csde
	12 and 345673	Khớp vì thỏa mãn điều kiện: 12 và 345673
	1 and 2	Không khớp vì chuỗi chỉ có 1 chữ số

Dấu sổ dọc |

Dấu sổ dọc | khớp với chuỗi tồn tại 1 trong 2 ký tự được định nghĩa trước và sau nó.
Ví dụ, a|b khớp với bất kỳ chuỗi nào chứa a hoặc b .

Biểu thức	Chuỗi ví dụ	Mô tả
a|b	cde	Không khớp vì a , b đều không xuất hiện
	ade	Khớp vì thỏa mãn điều kiện, có a xuất hiện: ade
	acdbea	Khớp vì thỏa mãn điều kiện, a và b đều xuất hiện: acdbea

Dấu ngoặc đơn ()

Dấu ngoặc đơn () sử dụng để gom nhóm các pattern lại với nhau, chuỗi sẽ khớp với biểu thức quy tắc bên trong dấu ngoặc.
Ví dụ: (a|b|c)xz khớp với bất kỳ chuỗi nào có a hoặc b hoặc c đứng trước xz .

Biểu thức	Chuỗi ví dụ	Mô tả
(a|b|c)xz	ab xz	Không khớp vì a hay b có đứng trước nhưng không liền với xz
	abxz	Khớp vì thỏa mãn điều kiện, có b xuất hiện sát trước xz : abxz
	axz cabxz	Khớp vì thỏa mãn điều kiện, cả a và b đều xuất hiện sát trước xz : axz cabxz

Dấu gạch chéo ngược \

Dấu gạch chéo ngược được sử dụng để thoát các ký tự đặc biệt, khi đứng trước kí tự đặc biệt, \ sẽ biến kí tự này thành kí tự thường.

Ví dụ: \$a khớp với chuỗi chứa ký tự $ đứng trước a

Biểu tượng $ không dùng để khớp chuỗi kết thúc bằng ký tự đi cùng, $ chỉ là ký tự bình thường.

Dấu gạch chéo ngược cũng biến kí tự thường liền kế phía sau thành kí tự đặc biệt.

Ví dụ, ký tự b khớp với ký tự b thường, nhưng \b trở thành kí tự đặc biệt, không khớp với bất kì ký tự nào.

Pattern đi với \

1. \A - Khớp với các ký tự theo sau nó nằm ở đầu chuỗi.

Biểu thức	Chuỗi ví dụ	Mô tả
\Athe	the sun	Khớp vì the nằm ở đầu chuỗi
	In the sun	Không khớp vì the không nằm ở đầu chuỗi

Pattern đi với \

2. \b - Khớp với các ký tự được chỉ định nằm ở đầu hoặc cuối của từ.

Biểu thức	Chuỗi ví dụ	Mô tả
\bfoo	football	Khớp vì thỏa mãn điều kiện, foo nằm ở đầu chuỗi
	a football	Khớp vì thỏa mãn điều kiện, foo nằm ở đầu từ thứ 2 trong chuỗi
	afootball	Không khớp vì foo nằm ở giữa từ trong chuỗi.
foo\b	the foo	Khớp vì thỏa mãn điều kiện, foo nằm ở cuối chuỗi
	the afoo test	Khớp vì thỏa mãn điều kiện, foo nằm ở cuối từ thứ 2 trong chuỗi
	the afootest	Không khớp vì foo nằm ở giữa từ trong chuỗi.

Pattern đi với \

3. \B - Trái ngược với \b , khớp với các ký tự được chỉ định không nằm ở đầu hoặc cuối của từ.

Biểu thức	Chuỗi ví dụ	Mô tả
\bfoo	football	Không khớp vì foo nằm ở đầu chuỗi
	a football	Không khớp vì foo nằm ở đầu từ thứ 2 trong chuỗi
	afootball	Khớp vì foo nằm ở giữa từ trong chuỗi.
foo\b	the foo	Không khớp vì foo nằm ở cuối chuỗi
	the afoo test	Không khớp vì foo nằm ở cuối từ thứ 2 trong chuỗi
	the afootest	Khớp vì foo nằm ở giữa từ trong chuỗi.

Pattern đi với \

4. \d - Khớp với các ký tự là chữ số, tương đương với [0-9] .

Biểu thức	Chuỗi ví dụ	Mô tả
\d	12abc3	Khớp vì thỏa mãn điều kiện: 12abc3
	Python	Không khớp vì không có số nguyên nào xuất hiện

Pattern đi với \

5. \D - Khớp với các ký tự không phải số, tương đương với [^0-9] .

Biểu thức	Chuỗi ví dụ	Mô tả
\D	1ab34"50	Khớp vì thỏa mãn điều kiện: 1ab34"50
	1345	Không khớp vì chuỗi toàn số nguyên xuất hiện

Pattern đi với \

6. \s - Khớp với bất kỳ ký tự khoảng trắng nào, tương đương với [\t\n\r\f\v] .

Biểu thức	Chuỗi ví dụ	Mô tả
\s	Python RegEx	Khớp vì chuỗi có khoảng trắng
	PythonRegEx	Không khớp vì chuỗi không có khoảng trắng

Pattern đi với \

7. \S - Khớp với bất kỳ ký tự nào không phải khoảng trắng, tương đương với [^ \t\n\r\f\v] .

Biểu thức	Chuỗi ví dụ	Mô tả
\S	a b	Khớp vì chuỗi có ký tự a b
		Không khớp vì chuỗi toàn bộ là khoảng trắng

Pattern đi với \

8. \w - Khớp với bất kỳ ký tự chữ cái và chữ số nào, tương đương với [a-zA-Z0-9_] .
Lưu ý: Dấu gạch dưới _ được coi là một ký tự chữ cái và chữ số.

Biểu thức	Chuỗi ví dụ	Mô tả
\w	12&": ;c	Khớp vì chuỗi có ký tự chữ và số 12&": ;c
	%"> !	Không khớp vì chuỗi không có ký tự chữ và số

Pattern đi với \

9. \W - Khớp với bất kỳ ký tự nào không phải là chữ cái và chữ số, tương đương với [^a-zA-Z0-9_].
Lưu ý: Dấu gạch dưới _ cũng được coi là một ký tự chữ cái và chữ số

Biểu thức	Chuỗi ví dụ	Mô tả
\w	1a2%c	Khớp vì chuỗi có ký tự không phải chữ và số 1a2%c
	Python	Không khớp vì chuỗi chỉ có ký tự chữ cái

Regular Expression

Để xây dựng biểu thức quy tắc RegEx, có thể sử dụng công cụ kiểm tra RegEx như regex101.

Công cụ này không chỉ tạo biểu thức quy tắc mà còn giúp tìm hiểu kỹ hơn.

Cách sử dụng RegEx.

Regular Expression trong Python

Regular Expression trong Python được thể hiện qua module re , muốn sử dụng regular expression cần import module re.

```
import re
```

Module này có rất nhiều các phương thức, hàm và hằng để làm việc với RegEx.

Một số hay được sử dụng:

re.findall()

Phương thức re.findall() trả về một danh sách các chuỗi chứa tất cả các kết quả khớp với pattern đưa ra.
Cú pháp:

```
>>> findall(partern, string)
```

Trong đó:
- pattern là RegEx.
- string là chuỗi cần so khớp.

Ví dụ: Trích xuất các số từ chuỗi cho trước sau: *"hello 12 hi 89. Howdy 34"*

re.findall()

```
>>> import re
    string = 'hello 12 hi 89. Howdy 34'
    pattern = '\d+'
    result = re.findall(pattern, string)
    print(result)
```

Kết quả:

```
>>> ['12', '89', '34']
```

re.split()

Phương thức re.split() dùng biểu thức quy tắc để ngắt chuỗi thành các chuỗi con và trả về danh sách các chuỗi con này.

Cú pháp:

```
>>> re.split(pattern, string, maxsplit)
```

Trong đó:
- pattern là RegEx.
- string là chuỗi cần so khớp.
- maxsplit (số nguyên) là số chuỗi tối đa sẽ được ngắt. Nếu để trống thì Python sẽ so khớp và cắt tất cả các chuỗi đạt điều kiện.

re.split()

Ví dụ: Ngắt tại vị trí có ký tự khoảng trắng:

```
>>> import re
    string = 'The rain in Vietnam.'
    pattern = '\s'
    result = re.split(pattern, string)
    print(result)
```

Kết quả: ['The', 'rain', 'in', 'Vietnam.']

Ví dụ: Ngắt chuỗi ở ký tự khoảng trắng đầu tiên:

```
>>> import re
    string = 'The rain in Vietnam.'
    pattern = '\s'
    result = re.split(pattern, string, 1)
    print(result)
```

re.split()

Kết quả:
>>> ['The', 'rain in Vietnam.']
Nếu không tìm thấy pattern, *re.split()* trả về danh sách chứa chuỗi rỗng.

re.sub()

Re.sub() sẽ thay thế tất cả các kết quả khớp với pattern trong chuỗi bằng nội dung khác được truyền vào và trả về chuỗi đã sửa đổi.

Cú pháp:

>>> re.sub(pattern, replace, string, count)

Trong đó:
- pattern là RegEx.
- replace là nội dung thay thế cho chuỗi khớp với pattern
- string là chuỗi cần so khớp.
- count (số nguyên) là số lần thay thế. Nếu để trống coi như giá trị này bằng 0, so khớp và thay thế tất cả các chuỗi đạt điều kiện

re.sub()

Ví dụ: Code chương trình xóa tất cả các khoảng trắng

```
>>> import re
    # chuỗi nhiều dòng
    string = 'abc 12\
              de 23 \n f45 6'
    # so khớp các ký tự khoảng trắng
    pattern = '\s+'
    # chuỗi rỗng
    replace = ''
    new_string = re.sub(pattern, replace, string)
    print(new_string)
```

Kết quả:

```
>>> abc12de23f456
```

Nếu không tìm thấy kết quả phù hợp với pattern, re.sub() sẽ trả về chuỗi rỗng.

re.sub()

Ví dụ: Code chương trình xóa 2 khoảng trắng đầu tiên

```
>>> import re
    # chuỗi nhiều dòng
    string = 'abc 12\
    de 23 \n f45 6 \n laptrinh nangcao'
    # so khớp các ký tự khoảng trắng
    pattern = '\s+'
    replace = ''
    new_string = re.sub(r'\s+', replace, string, 2)
    print(new_string)
```

Output trả về:

```
>>> abc12de23
     f45 6
     laptrinh nangcao
```

re.subn()

Tương tự **re.sub()**, nhưng kết quả trả về gồm tuple chứa hai giá trị: chuỗi mới sau khi được thay thế và số lần thay thế đã thực hiện.

```
>>> import re
    # chuỗi nhiều dòng
    string = 'abc 12\
    de 23 \n f45 6 \n laptrinh nangcao'
    # so khớp các ký tự khoảng trắng
    pattern = '\s+'
    # chuỗi rỗng
    replace = ''
    new_string = re.subn(pattern, replace, string)
    print(new_string)
```

Kết quả trả về:

```
>>> ('abc12de23f456laptrinhnangcao', 6)
```

re.search()

Phương thức re.search() sử dụng để tìm chuỗi phù hợp với pattern RegEx. Nếu tìm kiếm thành công, *re.search()* trả về đối tượng khớp, nếu không, trả về None.

Cú pháp:

```
>>> search(pattern, string)
```

Trong đó:
- pattern là RegEx.
- string là chuỗi cần so khớp.

```
>>> import re
    string = ‘Laptrinhnangcao la mon hoc quan trong’
    # Kiem tra ‘Laptrinhnangcao' co nam o dau chuoi
    match = re.search('\ALaptrinhnangcao', string)
    if match: # nếu tồn tại chuỗi khớp
        print("Tim thay ‘Laptrinhnangcao' nam o dau
chuoi") # in thông báo này
    else:
        print("‘Laptrinhnangcao' khong nam o dau
chuoi") # khong thì in thông báo náy
```

Kết quả:

```
>>> Tim thay ‘Laptrinhnangcao' nam o dau chuoi
```

Ở ví dụ này, match chứa đối tượng khớp với pattern.

Đối tượng match

match.group() trả về những phần của chuỗi khớp với pattern.

```
>>> import re
    string = '39801 356, 2102 1111'
    pattern = '(\d{3}) (\d{2})'
    match = re.search(pattern, string)
    if match: #nếu tồn tại chuỗi khớp
       print(match.group()) # in ra kết quả
    else:
      print("Không khớp") # Không thì hiện thông báo
# Output: 801 35
```

Đối tượng match

biến match chứa đối tượng match.
pattern (\d{3}) (\d{2}) chia làm hai nhóm nhỏ (\d{3}) và (\d{2}), nhận một phần của chuỗi tương ứng với các nhóm con trong ngoặc đơn:

```
>>> match.group(1)
    '801'
>>> match.group(2)
    '35'
>>> match.group(1, 2)
    ('801', '35')
>>> match.groups()
    ('801', '35')
```

.start(),.end() và.span()

Hàm start() trả về chỉ mục bắt đầu của chuỗi con phù hợp. Tương tự, **end()** trả về chỉ mục kết thúc của chuỗi con phù hợp.

```
>>> match.start()
    2
>>> match.end()
    8
```

Hàm span() trả về tuple chứa chỉ mục bắt đầu và kết thúc của phần chuỗi phù hợp

```
>>> match.span()
    (2, 8)
```

match.re và match.string

Thuộc tính re của đối tượng match sẽ trả về một biểu thức quy tắc. Tương tự, thuộc tính string trả về chuỗi đã được truyền trong đoạn code.

```
>>> match.re
    re.compile('(\\d{3}) (\\d{2})')
>>> match.string
    '39801 356, 2102 1111'
```

Trên đây là tất cả các phương thức thường được sử dụng nhất trong module re.

Sử dụng tiền tố r trước RegEx

Tiền tố r hoặc R sử dụng trước biểu thức RegEX đại diện cho chuỗi theo sau là những ký tự bình thường.

Ví dụ: '\n' là một dòng mới, còn r'\n' có nghĩa là chuỗi bao gồm hai ký tự: dấu gạch chéo ngược \ và n .

Dấu gạch chéo ngược \ được sử dụng để thoát các ký tự đặc biệt. Tuy nhiên, sử dụng tiền tố r trước \ thì là ký tự bình thường.

```
>>> import re
        string = '\n and \r are escape sequences.'
        result = re.findall(r'[\n\r]', string)
        print(result)
>>> # Output: ['\n', '\r']
```

Bài tập

1. Xác định hàm với generator lặp lại các số nằm trong khoảng 0 và n, và chia hết cho 7.
Gợi ý: Sử dụng yield.

2. Định nghĩa một class có tên là Vietnam, với static method là printNationality. (**Dùng @staticmethod**)

3. Viết chương trình sử dụng generator để in số chẵn trong khoảng từ 0 đến n, cách nhau bởi dấu phẩy, n là số được nhập vào. (Sử dụng yield)

4. Viết chương trình sử dụng generator để in số chia hết cho 5 và 7 giữa 0 và n, cách nhau bằng dấu phẩy, n được người dùng nhập vào. (Sử dụng yield)

Bài tập

5. Một website yêu cầu nhập tên người dùng và mật khẩu để đăng ký. Viết chương trình để kiểm tra tính hợp lệ của mật khẩu mà người dùng nhập vào. Các tiêu chí kiểm tra mật khẩu bao gồm:

1. Ít nhất 1 chữ cái nằm trong [a-z]
2. Ít nhất 1 số nằm trong [0-9]
3. Ít nhất 1 kí tự nằm trong [A-Z]
4. Ít nhất 1 ký tự nằm trong [$ # @]
5. Độ dài mật khẩu tối thiểu: 6
6. Độ dài mật khẩu tối đa: 12

(**import re**)

Bài tập

Chương trình phải chấp nhận một chuỗi mật khẩu phân tách nhau bởi dấu phẩy, kiểm tra xem chúng có đáp ứng những tiêu chí trên hay không. Mật khẩu hợp lệ sẽ được in, mỗi mật khẩu cách nhau bởi dấu phẩy.
Ví dụ mật khẩu nhập vào chương trình là:
ABd1234@1,aF1#,2w3E*,2We3345
Thì đầu ra sẽ là: ABd1234@1

Bài tập

6. Giả sử địa chỉ email dạng username@companyname.com, hãy viết một chương trình để in username của địa chỉ email cụ thể. Cả username và companyname chỉ bao gồm chữ cái

Sử dụng \w để kiểm tra chữ cái (**import re**)

7. Tương tự như bài 6, viết hàm để lấy companyname.(**import re, re.match(), .group()**)

8. Viết một chương trình nhận chuỗi từ được phân tách bằng khoảng trống và in các từ chỉ gồm chữ số

Sử dụng re.findall()