ITERATORS AND GENERATORS

CS 3030: Python

Instructor: Damià Fuentes Escoté

University of Colorado
Colorado Springs

Previous lesson

- Object-oriented programming
- Classes
- Instances
- Class and instance attributes
- Instantiating objects
- Instance methods
- Inheritance
- Encapsulation
- Polymorphism

Iterators

- We use **for** statement for looping over a **list**.

```
>>> for i in [1, 2, 3, 4]:
...     print(i)
...
1
2
3
4
```

Iterators

- If we use it with a **string**, it loops over its characters.

```
>>> for c in "python":
...     print(c)
...
p
y
t
h
o
n
```

Iterators

- If we use it with a **dictionary**, it loops over its keys.

```
>>> for k in {"x": 1, "y": 2}:
...     print(k)
...
x
y
```

Iterators

- If we use it with a **file**, it loops over lines of the file.

```
>>> for line in open("a.txt"):
...     print(line)
...
first line
second line
```

We will see this in 3 classes.

Iterators

- So there are many types of objects which can be used with a for loop.
- These are called **iterable objects**.

Iteration protocol

- The built-in function **iter** takes an iterable object and returns an iterator.

```
x = iter([1, 2, 3])
print(type(x))    # <class 'list_iterator'>
print(next(x))    # 1
print(next(x))    # 2
print(next(x))    # 3
print(next(x))    # Traceback (most recent call last):
                  #   File "<stdin>", line 1, in <module>
                  # StopIteration
```

Creating an iter object

- To make a custom class be iterable, it has to implement the **__iter__** and **__next__** methods.
  - *The **__iter__** method is what makes an object iterable. The return value of **__iter__** is the class itself.*
  - *The __**next__** method is what the class should return at each iteration. It raises **StopIteration** when there are no more elements.*

Creating an iter object

```python
class MyRange:
    def __init__(self, n):
        self.i = 0
        self.n = n

    def __iter__(self):
        return self

    def __next__(self):
        if self.i < self.n:
            result = self.i
            self.i += 1
            return result
        else:
            raise StopIteration()
```

Generators

- Generators simplifies creation of iterators. A generator is a function that produces a sequence of results instead of a single value.
- Each time the **yield** statement is executed the function generates a new value.
- When a generator function is called, it returns a generator object without even beginning execution of the function. When next method is called for the first time, the function starts executing until it reaches yield statement. The yielded value is returned by the next call.

```
def myRange(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

Generators examples

```
def integers():
    """Infinite sequence of integers."""
    i = 1
    while True:
        yield i
        i = i + 1

def squares():
    """Infinite sequence of integer squares."""
    for i in integers():
        yield i * i
```

Generators examples

```
def take(n, seq):
    """Returns first n values from the given sequence."""”
    seq = iter(seq)                 # Just in case it is an iterable object,
                                    # not a generator or iterator
    result = []
    try:
        for i in range(n):
            result.append(next(seq))
    except StopIteration:
        pass
    return result

print(take(5, squares())) # [1, 4, 9, 16, 25]
```

List comprehensions

From the Lists class

- List comprehension is an elegant way to define and create list in Python.

```
my_list = [x * 2 for x in range(10)]

print(my_list)
```

List comprehensions

From the Lists class

- List comprehension is an elegant way to define and create list in Python.

```
my_list = [x * 2 for x in range(10)]

print(my_list)    # [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Generators expressions

- Generator expressions allow the creation of a generator on-the-fly without a yield keyword.
- They look like list comprehensions, but returns a generator back instead of a list.
- In terms of syntax, the only difference is that you use parenthesis instead of square brackets.

```
gen_exp = (x ** 2 for x in range(10) if x % 2 == 0)


print(type(gen_exp))        # <class 'generator'>
for x in gen_exp:
     print(x)               # 0
                            # 4
                            # 16
                            # 36
                            # 64
```

Generators expressions

- The type of data returned by list comprehensions and generator expressions differs.

```
list_comp = [x ** 2 for x in range(10) if x % 2 == 0]

gen_exp = (x ** 2 for x in range(10) if x % 2 == 0)

print(list_comp)
# [0, 4, 16, 36, 64]
print(gen_exp)
# <generator object <genexpr> at 0x7f600131c410>
```

Generators expressions

- The main advantage of generator over a list is that it take much less memory.
- The generator yields one item at a time—thus it is more memory efficient than a list.

```
from sys import getsizeof

my_comp = [x * 5 for x in range(1000)]
my_gen = (x * 5 for x in range(1000))

print(getsizeof(my_comp))
# 9024
print(getsizeof(my_gen))
# 88
```

Time to code – Reverse iterator - HW4 ex2

- Write an iterator class ReverseIter, that takes a list and iterates it from the reverse direction.

```
myRange = ReverseIter([1, 2, 3, 4])
print(next(myRange))
print(next(myRange))
print(next(myRange))
print(next(myRange))
# 4
# 3
# 2
# 1
```

Time to code – Reverse iterator - HW4 ex3

- Use a **generator comprehension** expression to find first 10 (or any n) pythogorian triplets.
- A triplet (x, y, z) is called pythogorian triplet if
  - *x*x + y*y == z*z.*

```
pyt = ((x, y, z) ... Generator comprehension ...)
print(take(10, pyt))
# [(3, 4, 5), (6, 8, 10), (5, 12, 13), (9, 12, 15), (8, 15,
17), (12, 16, 20), (15, 20, 25), (7, 24, 25), (10, 24, 26),
(20, 21, 29)]
```

Time to code – The Generator Version of range() - HW4 ex4

- In past versions of python (like python 2.7), the range(x) function generated a list.
  - *In python 3 the range() function is optimized. Something similar to what we are going to do.*

- Define a generator, genrange(), which generates the same sequence of values as range(), without creating a list object.