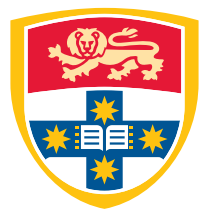
INFO1110 & COMP9001: Introduction to Programming

School of Information Technologies, University of Sydney



Copyright Warning

Week 10: Functional programming

We will cover: Iterators, Generators, and the for loop

You should read: §§pg461-468, 478, 661

Lecture 19: Functional programming

Iterators

Iterators

Representing a stream of data

Iterators

An iterator is an object representing a stream of data.

The stream of data could be finite, or infinite.

The iterator has two fundamental operations:

- return the next element in the stream of data
- test if reached the end of the stream

Iterator examples

Given a collection, traverse through each item:

- in the order they are stored
- in sorted ascending order
- in order of addition
- in order of importance
- selectively
- in random order

Collection may be:

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

Existing Iterator: list

You have already been using iterators. List objects support the iterator functionality.

```
1 strings = ["Do", "we", "sell...", "French...", "fries?" ]
2 p = iter(strings)
3 print(type(p))
4 print(p.__next__())
5 print(p.__next__())
6 print(p.__next__())
7 print(p.__next__())
8 print(p.__next__())
9 print(p.__next__())
```

Existing Iterator: list (cont.)

```
$ python3 iterator_test.py
<class 'list_iterator'>
Do
we
sell...
French...
fries?
Traceback (most recent call last):
  File "iterator_test.py", line 9, in <module>
    print(p.__next__())
StopIteration
```

Iterator class vs Iterator object

Suppose we have a collection of items that we wish to traverse in a certain order.

- The iterator class represents how the traversal is performed
- The iterator object represents the current state of the traversal (position of the cursor)

Make your own Iterator

Example. We want to create an iterator for a collection of people's names. The iterator would give us the same sequence of names as they are presented in the collection.

We first need to define this new data type for a collection of names

```
1 class People:
2     def __init__(self, names):
3         self.names = names
```

Iterator requirement 1: The data type has to support a method `__iter__()` to return an iterator object

```
1 class People:
2     def __init__(self, names):
3         self.names = names
4 
5     def __iter__(self):
6         return PeopleIterator(self.names) # what is this?!
```

Make your own Iterator (cont.)

`PeopleIterator` is our iterator object, we need a new class

Iterator requirement 2: The iterator object is also new data type and supports the `__next__()` method

```
1  class PeopleIterator:
2      def __init__(self, names):
3          # initialised with some data necessary for the iterator
4          self.names = names
5
6      def __next__(self):
7          # returns the next name in the collection
8          return ???
```

Make your own Iterator (cont.)

Iterator requirement 3: The iterator class needs to define how the iterator state is updated, what element is to be returned on each call to `__next__()` method and raise the exception `StopIteration`

Make your own Iterator (cont.)

```
1  class PeopleIterator:
2      def __init__(self, names):
3          self.names = names
4          # the state of the iterator is defined as the index of the
               list
5          self.cursor = 0
6
7      def __next__(self):
8          # if the end of the list reached, raise exception
9          if self.cursor >= len(self.names):
10             raise StopIteration("Reached end")
11
12         # get the next element to return
13         ret_val = self.names[self.cursor]
14
15         # update the iterator state to the next element
16         self.cursor += 1
17
18         # return the value
19         return ret_val
```

Make your own Iterator (cont.)

```
class People:
  def __init__(self, names):
    self.names = names

  def __iter__(self):
    return PeopleIterator(self.names)
```

```
class PeopleIterator:
  def __init__(self, names):
    self.cursor = 0
    self.names = names

  def __next__(self):
    if self.cursor >= len(self.names):
      raise StopIteration("Reached end")
    ret_val = self.names[self.cursor]
    self.cursor += 1
    return ret_val
```

```
simpsons = People(["Armen", "Hans", "Lunchlady Doris"])
myiterator = iter(simpsons)
print(myiterator.__next__())
print(myiterator.__next__())
print(myiterator.__next__())
print(myiterator.__next__())
```

Make your own Iterator (cont.)

```
$ python3 IteratorPeople.py
Armen
Hans
Lunchlady Doris
Traceback (most recent call last):
  File "IteratorPeople.py", line 25, in <module>
    print(myiterator.__next__())
  File "IteratorPeople.py", line 15, in __next__
    raise StopIteration("Reached end")
StopIteration: Reached end
```

Make your own: A specialised traversal

Exercise. A Zoo contains many animals. Each animal has a name, an age, and an animal type. Create an iterator for the Zoo object. The iterator will return a string upon each call to `next()`. The returned element is a string that combines the three features of an Animal for presentation to children:

```
<Name> the <animal type> is <age> years old
```

The ordering of the elements in the iterator is based on the ascending order for names of the animal e.g. "Eric the Hippo" followed by "Garrison the Bear"

The classes `Animal` and `Zoo` are provided to you

Make your own: A specialised traversal (cont.)

Classes provided

```
1  class Animal:
2      def __init__(self, name, age, a_type):
3          self.name = name
4          self.age = age
5          self.a_type = a_type
6
7      def get_name():
8          return self.name
```

```
1  class Zoo:
2      '''contains a list of Animal objects'''
3      def __init__(self, animals):
4          self.animals = animals
```

```
1  zoo = initialise_zoo()
2  names = iter(zoo)
3  print(names)
```

iterators are one way to define an iterable object

Have you been using for loops and don't know how it works?

The for loop in Python is complicated! There are many hidden operations

Here is an example for loop

```
1 text = "You're in the newspaper business?"
2 for ch in text:
3     print(ch, end='')
4
5 print('')
```

Here is the equivalent using while loop

iterators are one way to define an iterable object (cont.)

```
1  text = "You're in the newspaper business?"
2  text_iterator = text.__iter__()
3  while True:
4      try:
5          ch = text_iterator.__next__()
6          print(ch, end='')
7      except StopIteration:
8          break
9
10 print('')
```