CSc 120
Introduction to Computer Programming II

16: Python extras
lambda expressions

• Anonymous functions can be made with the lambda keyword
  – lambda a, b : a + b

• This is a syntactic short hand

• Restricted to single expressions

• Lambda functions can be used wherever a function object is valid
lambda expressions

• Example:
>>> def make_incrementer(n):
    return lambda x: x + n

>>> f = make_incrementer(10)
>>> f
<function make_incrementer.<locals>.<lambda> at 0x10570bf28>
>>> f(2)
12
>>> f(100)
110
>>>
Iterators

• Iterator:
  – a Python object that can be iterated upon
  – an object that returns data, one element at a time

• Container types (lists, tuples, strings, etc.) are iterables

• Example use in a for loop:
  >>> for elem in "abcde":
      print(elem)

• The for loop automatically gets each element of an iterator in sequence
Iterators

• iterators can be used outside of for loops
• an iterator implements two special methods
  
  __ iter__ () : returns an iterator object

  __ next__ () : produces the next value in the iterator sequence
  raises StopIteration when finished

• the above is the \textit{iterator protocol}
Iterators: Example

```python
>>> alist = [2, 4, 6, 8]

>>> x = iter(alist)  # create an iterator

>>> next(x)          # use next to get the next element in the sequence
2

>>> next(x)
4

>>> next(x)
6

>>> next(x)
8

>>> next(x)
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    next(x)
StopIteration
```

6
the for loop

• the for loop uses iterables
  for element in iterable:
    # do something with iterable

• implementation
  iter_obj = iter(iterable)

  # infinite loop
  while True:
    try:
      element = next(iter_obj)
    except StopIteration:
      break
user-defined iterators

• must define the method `__next__()`
  returns the next value in the sequence

• must define the method `__iter__()`
  returns an object that has a `next()` method
  just return `self`

• let's write an iterator that reverses a sequence of values
user-defined iterators

# an iterator to reverse a sequence of values

class Reverse:
    def __init__(self, data):
        self._data = data
        self._index = len(data)  # start the index at the end

    def __iter__(self):
        return self  # define the iterator

    def __next__(self):
        if self._index == 0:
            raise StopIteration
        self._index = self._index - 1
        return self._data[self._index]  # return the next value in the sequence
user-defined iterators

```python
>>> rev = Reverse("aeiou")
>>> next(rev)
'u'
>>> next(rev)
'o'
>>> for c in rev:
    print(c)

i
e
ea
```
Generators

• generators are a syntactic mechanism for creating iterators

• generator
  – written like a function
  – uses yield instead of return

• each time next() is called, the generator resumes where it left off
Generators

• the iter() and next() methods are created automatically

• the generator version of reverse:
  ```python
def reverse_gen(data):
    for index in range(len(data)-1, -1, -1):
      yield data[index]
  ```
Generators

>>> def reverse_gen(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

>>> reverse_gen
<function reverse_gen at 0x101183510>

>>> for char in reverse_gen("abcde"):
    print(char)

e
d
c
b
a

>>>