CSc 120
Introduction to Computer Programming II

15: Python extras

lambda expressions

- Anonymous functions can be made with the lambda keyword
- Normally we define the name of a function:

```
def double(x):
    return x * 2
```

- Anonymous version:

```
lambda x: x * 2
```

- This is a syntactic shorthand in Python
- Restricted to single expressions

lambda expressions

- Origin is the lambda calculus (Church, 1936-40)
  - The basis of functional programming languages
  - Take 372! (Comparative Programming Languages)
- Lambda functions can be used wherever a function object is valid
- Functions are 1st class objects
  - can be passed as arguments
  - can be return as results

lambda expressions

- Example:

```
>>> def make_incrementer(n):
        return lambda x: x + n

>>> f = make_incrementer(10)          # lambda x: x + 10
>>> type(f)
<class 'function'>
>>> f(2)
12
>>> f(100)
110
>>>
```

lambda expressions

- Example:

```
>>> g = make_incrementer(1)
>>>
>>> g
<function make_incrementer.<locals>.<lambda> at 0x7f8c72119310>
>>> g(2)
3
>>> g(100)
101
>>>
```

lambda expressions

- Lambda functions are often used with the following built-in functions
  - filter
  - map
  - reduce
  - sort

- We'll look at filter:
  - `filter(function, sequence)`
    - applies `function` to each element of `sequence`

filter

- filter is used to filter out elements of a sequence
- `filter(f, sequence)`
  - Takes a function `f` and a sequence `sequence`
  - the function `f` returns a boolean
  - the function `f` is applied to every element of `sequence`; if it returns True, the element is added to the result sequence
  - returns a filter object
  - can make this a list

filter

- Example: Filter out the even numbers

```
>>> a   = [2,  7,  9,  32,  6,  14]

>>> result  =  list(filter(lambda x: x % 2 ==  0,  a))
>>> result
[2,  32,  6,  14]
>>>
```

EXERCISE

```
>>> alist = ["able", "dig", "byte", "eye"]
>>> list(filter(lambda w: len(w) % 2 == 1, alist))
```

what do you think will be printed here?

EXERCISE-whiteboard

```
>>> alist = ["able", "dig", "byte", "eye"]
>>>
```

Use filter on alist to produce the list of words that start with a vowel, i.e.,

```
['able', 'eye']
```

Prior example:

```
list(filter(lambda w: len(w) % 2 == 1, alist))
```

EXERCISE-SOL

```
>>>alist = ["able", "dig", "byte", "eye"]
>>>list(filter(lambda w: w[0] in "aeiou", alist)
['able', 'eye']
```

Iterators

- a for loop iterates over a sequence of values:

```
for elem in "abcde "
    print(elem)
```

- works for any sequence of values
  - lists, strings, dictionaries, tuples, sets
  - any container type – list, dictionary, tuple, set
- the for statement calls a function iter() on the sequence type
- the iter() function creates and returns an iterator object

Iterators

- Iterator:
  - a Python object that can be iterated upon
  - an object that returns data, one element at a time
  - when there are no more elements, raises a StopIteration exception

Iterators

- the iter() method creates an iterator object
- can accept any type that has a sequence of values
- 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 *iterator protocol*
- iterators can be called outside of for loops

Iterators: Example

```
>>> 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
>>>
```

EXERCISE-whiteboard

```
>>> romans = {"I": 1, "V": 5, "X": 10, "L": 50}
```

Create an iterable object from the dictionary romans. Using next(), print the first two keys in the dictionary, one at a time.

the for loop

- the for loop can be used for any type that has a sequence of values (an iterable)

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

- implementation

```
iter_obj = iter(iterable)

# infinite loop
while True:
    try:
        element = next(iter_obj)
         # do something with element
    except StopIteration:
        break
```

the for loop

- the for loop can be used for any type that has a sequence of values (an iterable)

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

- implementation

```
iter_obj = iter(iterable)

# infinite loop
while True:
    try:
        element = next(iter_obj)
       # do something with element
    except StopIteration:
        break
```

Use of try/except!

user-defined iterators

- define a class for the iterator
- 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
- Ex: let's write an iterator that reverses a sequence of values called Reverse()
- Let’s see how to use it first

user-defined iterators

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

i
e
a
>>>
```

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 and
                                             # decrement

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

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

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:

```
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
>>>
```

Python Language Design

- Python was designed by Guido Van Rossum
  - Developed in the mid 80's
  - Primary influence was the language ABC (was used for teaching/prototyping)
  - Guido also acknowledges the influence of Icon
  - **The Icon Programming Language:**
  - Designed by Ralph Griswold (in our CS dept.!)
  - Has generators
  - The language has implicit backtracking
    - (We'll see backtracking Wednesday)

EXERCISE-ICA-41

Do all problems.