INFO1110 / COMP9001 Week 10 Tutorial

Functional programming

What is functional?

"Decomposition of the problem into a set of functions that do not have an internal state.". This approach commonly employs immutability to variables and the removal of state being used in functions as a way of coding. Although python supports this paradigm, it not necessarily the dominant one. Python is much more heavily influenced by imperative and object oriented paradigms.

Function variables and callbacks

On top of having variables that can be bound to all kinds of data we can also have variables bound to functions and pass these functions as parameters.

Lambda Functions

We can think of lambda functions as anonymous functions, these functions are typically short and bound to a variable or passed into another function.

```
fn = lambda x: x + 1
```

Iterators

An iterator is used to maintain the state and position while iterating through a collection or generating a set of values. We can create our own iterators in a similar manner to filter and range where we return the next value on the next function call.

We can extract an iterable object like so:

```
i = iter([1, 2, 3, 4])
print(next(i))
#usage in for loop
for k in i:
  print(k)
```

- How do we build our own?

We will do this by overloading the `__iter__` method on the class containing the elements. We will need to write a class that has access the source instance and will overload the `__next__` method.

Example of iterator type

```
import random

class RandomIterator:
  def __init__(self, n, seed):
    random.seed(seed)
    self.n = n
    self.i = 0

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

Builtin functions

`filter()`

`filter` is a predicate, the items that are returned are those that meet the criteria specified. When we execute `filter`, `map` or `zip` it will return an iterator object. By wrapping list on that iterator object, it will move through each element and add them to a list

```
l = [1, 2, 3, 4]
a = list(filter(lambda x: x % 2, l))
print(a) #[1, 3]
```

`map()`

`map` applies a function to all elements in a collection, this method will return an iterator that allow you move through collection returned, however we can also (not excluding filter) mutably change elements within the collection.

```
l = [1, 2, 3, 4, 5]
p = list(map(lambda x: x+1, l))
print(p) #[2, 3, 4, 5, 6]
```

zip

zip allows multiple collections to be merged into one collection, once the zip class is executed it will return an iterator which will allow you to traverse it. The data sets given, takes an element from each list and assembles a tuple.

Breakdown:

```
l = [1, 2, 3, 4]
k = [9, 8, 7, 6]

=> f = zip(l, k)
# what each element from the iterator will expand to
f[0] = (1, 9)
f[1] = (2, 8)
f[2] = (3, 7)
f[3] = (4, 6)

p = list([(1, 9), (2, 8), (3, 7), (4, 6)])
```

Question 1: Filter all strings that start with jo

You will need to use the filter function to retrieve all elements that start with jo

Given the following list of strings:

```
l = ['Jumpy', 'Jolly', 'Jolting', 'Jimmys']
```

```
Jolly
Jolting
```

Hint: Use the filter function in conjunction with the list and lambda function

Question 2: Sort with lowercase

Write a lambda function that will sort a list of strings and ignore their case. You can attach a lambda function to key attributes in the sort function.

Example of using a lambda function with sort

```
l = [1, 2, 3, 4]
l.sort(key=lambda x: -x)
```

Extension: Write a lambda function that will sort all string elements by their reversed representation. Hint: Use string slices

Question 3: Extract all even numbers from a given set

You are tasked with writing a lambda function that will return a list of elements which are all even.

Use the `filter` function to test your query

```
l = [2, 4, 7, 9, 3, 6, 9, 6, 5, 3, 7, 2]
fn = filter(lambda _: __, l)
```

Example of a lambda function

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

Extension: Change this lambda expression to retrieve every element where $x \ mod \ n$ is 0.

Hint: Try an inner lambda expression.

Generators

We will introduce the `yield` keyword into python. This is not a commonly used keyword but it comes with some really exciting and useful cases and is in particular useful with generators.

There is a distinct difference between generators and iterators. Iterators require the implementation of `__iter__` and `__next__` on the class object while a generate is a function that utilises the `yield` keyword. Generators will half execution of a function or can be seen as such while iterators are a mechanism of moving through a set of data.

The yield keyword will halt the current functions current execution and return the value on its right hand side (similar to a return statement). When the function is executed again it will then resume execution from when it yielded. You can use a generator function in a for loop in the same manner as an iterator.

The yield keyword will return a generator object when first called, to call it outside of a for loop we will need use the `next` function which will subsequently call the `__next__` function on the generator object to resume execution until the next yield statement.

```
def gen_animals():
        yield "Dog"
        yield "Rabbit"
        yield "Cat"
        yield "Horse"
        yield "Duck"

for a in gen_animals():
        print(a)

g = gen_animals()
print(next(g))
print(next(g))
...
```

Question 4: Rewrite the range function using the yield

Using the yield keyword, you will rewrite the range function (call it `m_range`) that will return a number within the range specified. This will operate similar to the `range` function you have used previously.

The function signature:

```python
def m_range(end):
```

When used within a loop, `m_range` will grab the next value in the series start. It will return integers within $[0, n)$

```python
for n in m_range(10):
```

Extension: Change `m_range` so it can support an optional starting value if specified.

```python
def m_range(start, end=None):
```

Question 5: Building our own data structure

Given that you know about the overloading of the `__next__` and `__iter__` methods associated with classes. Create your own list class called `TypedList` which will only one type to exist within the collection.

It must have these properties:

- Provides the type of the list, to ensure that all elements of this list is of a certain type.
- Checks that any elements added will be of the same type, rejected if they are not.
- `append` and `prepend` methods for adding elements at `size` position or index `0` respectively.
- Iterators to be used so it can be used in a for loop like a regular list.

The scaffold is on the next page

```python
class TypedList:
  def __init__(self, type):
    """
    Initialises the class
    """
    pass

  def append(self, element):
    """
    Appends an element to the end of the list
    """
    pass

  def prepend(self, element):
    """
    Prepends an element to the start of the list
    """
    pass

  def insert(self, index, element):
    """
    Inserts at index and shifts all elements
    from index to n by 1
    """
    pass

  def get(self, index):
    """
    Retrieves an element from a specified index
    """
    pass

  def remove(self, index):
    """
    Removes an element from a specified index,
    you will need to
    shift all elements down
    """
    pass
```

You will need to use the `isinstance` method for this class.

Extension: Overload the `__getitem__` operator so it can be used in a similar manner as a list.

Question 6: Factorial Generator

You are to write a generator function that will generate the next factorial number in the series. Your generator function will have a number of steps they will need to execute before ending.

Function prototype:

```python
def factorial(n):
```

Example of usage:

```python
fn = factorial(4)
print(next(fn)) #1
print(next(fn)) #2
print(next(fn)) #6
print(next(fn)) #24
```

Discussion

- What is the difference between a `lambda` function and `regular` function in python?
- Can we use these functions interchangeably, what are the pros and cons?
- Can we have multiple statements with a lambda function?
- When would you use the zip function? If you were to rewrite the zip function how would you do this? **Extension: Write a zip function**
- What would happen if we give uneven collections to zip?
- What is the difference between an iterator and a generator?
- When would we use one over the other?
- What would happen if we mixed yield and return within a function?
- What other ways could we implement the `__iter__` method that may not require the use of a `__next__` function?

References

Functional Programming HOWTO

Origins of Python's "Functional" Features, Guido van Rossum