THE UNIVERSITY OF SYDNEY

INFO1910 S2 2023 Week 11 Tutorial

Iterators and Generators

Exceptions

When an error occurs, Python will throw an exception. These exceptions can be caught and handled, preventing the code from crashing.

```
try:
  5 / 0
except ZeroDivisionError:
  print("Oops")
```

This is performed using a `setjmp` operation and overwriting a crash in the program to return to the state of the set jump, then continuing with the exception block.

From a style point of view, exceptions should be a matter of last resort, proper input formatting is always a better approach than resorting to a very crude nominally undeclared and widely scoped GOTO. This only becomes messier if functions are included within the try block.

It's a good, if not explicitly necessary idea to specify what error is being caught in each case. While you can leave this blank, it generally demonstrates that the programmer does not quite know what their code is doing or why it is crashing.

An exception can be manually raised using the `raise` keyword.

```
raise TypeError
```

Exceptions are also objects in Python and you can create your own by inheriting from the `Exception` class.

There are many builtin concrete exceptions within Python that you can find within the documentation. It is generally advised to use an appropriate and meaningfully named exception for the type of error you are handling.

This still leaves the question on how on earth an Exception can be built. It clearly violates normal control flow conventions, and leaves open the question of whether or not the block rolls back. Once again we can turn to an approximate C implementation:

```c
#include <stdio.h>
#include <setjmp.h>
#define JUMP_FLAG (2)

int main()
{
    jmp_buf saved_registers;

    // TRY
    if (setjmp(saved_registers) != JUMP_FLAG)
    {
        int val;
        scanf("%d", &val);

        if (0 == v)
        {
            longjmp(saved_registers, JUMP_FLAG);
        }
        val = 1 / val;
    }
    else // EXCEPT
    {
        printf("Divsion by 0!");
    }
    return 0;
}
```

Question 1: Rollback

Given the above C code predict the value of `y`.

```python
try:
    x = int(input())
    y = 5
    x = 1 / x
except:
    print("Error!")
```

Nested Exceptions

We can extend our C approximate by placing each saved `setjmp` on a stack (a list where we only ever place at the end and remove from the end). This ensures that we only ever return to the most recent nested exception. When we resolve a '`longjmp`' we can use the flag to indicate what the 'type' of the error was, which allows us to select on the exception type.

```
try:
    try:
        x = int(input())
        y = 5
        x = 1 / x
    except ZeroDivisionError:
        print("Divided by Zero!")
except ValueError:
    print("Input was not an integer!")
```

Of course in this case it is far better to compose exceptions than it is to nest them. And in general exceptions should be as limited as possible.

```
try:
    x = int(input())
except ValueError:
    print("Input was not an integer!")
    return None
try:
    x = 1 / x
except ZeroDivisionError:
    print("Divided by Zero!")
    return None
return x
```

Recursion

A recursive function is one that calls itself. We have seen this idea before and it also applies in Python.

Python is a applies an arbitrary limit on recursive depth, by default the recursive limit is 1000. It is normally suggested to find a way to implement a loop based solution rather than a recursive solution where possible.

Question 2: A new type of endlessness

Observe what the following code will do:

```
def rec(a):
    return rec(a - 1)
```

What happens when you try to run it?

What about this code?

```
def rec(a):
    if a < 2:
```

```
        return a
    else:
        return rec(a - 1) + rec(a - 2)
```

Try running this code for a few low values (less than 10) and then try a higher value (such as 30 or 40). What can you say about the performance of this code?

Question 3: Factorial

Write a recursive factorial function that will compute a factorial number of N.

What is your base case for a factorial function?

```
def rec_factorial(n)
```

Execute your code using the following statements:

```
print(rec_factorial(2)) # 2
print(rec_factorial(3)) # 6
print(rec_factorial(4)) # 24
```

What do you observe if you execute this:

```
print(rec_factorial(1001)) # 24
```

Question 4: Binomial Coefficients

Using your factorial function from the previous question. Write a binomial coefficient function (commonly referred to as an **n choose k** function).

Implement it with the following function prototype:

```
def binomial_coefficient(n, k)
```

A binomial coefficient is defined as:

$$\frac{n!}{k!(n-k)!}$$

Question 5: Fibonacci

Write a Python function that prints the n^{th} Fibonnaci number with recursion.

```
def rec_fibonacci(n)
```

The fibonacci recurrence relation is defined as:

$$F_n = F_{n-1} + F_{n-2}$$

The fibonacci sequence is as follows: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...$

You will need to consider the first two initial values of this sequence for your base case.

Question 6: Hailstones

Implement the Hailstone problem recursively, record all the numbers it computes to reach 1.

```
def rec_hailstone(n)
```

- If the number is even divide it by two
- If the number is odd multiply it by three and add one.

Test your code with the following statements:

```
print(hailstone(5)) # [5, 16, 8, 4, 2, 1]
```

Question 7: Loop Fibonacci

Convert your recursive fibonacci function to use loops. This is known as an iterative method and you may find it more complex than the recursive method.

```
def fibonacci_iterative(n)
```

- What complexities are associated with this method?
- What are the benefits of implementing it recursive over iteratively?
- Why would we still prefer to implement it iteratively than recursively?

Question 8: Flatten

You are given a Python object consisting of a collection of collections of collecitons ... of objects.

You are to write a recursive function that will 'flatten' this object into it's non collection objects.

For example:

```
x = [[1,2],3,[[4],[5,6,[7,8],[9]], 10], 11]
flatten(x)
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
```

Your numbers may be printed in a different order depending on how you implemented this problem.

Question 9: Tower of Hanoi (Extension)

Tower of Hanoi is a puzzle invented by E. Lucas in 1883. It consists of three rods, and a number of disks of different sizes arranged from largest on the bottom to smallest on the top placed on a rod. The objective of the puzzle is to take minimum number of steps to move the entire stack to another rod with following rules:

- Only one disk can be moved at a time.
- Cannot place a larger disk onto a smaller disk.
- A disk can only be moved if it is the top of the stack.

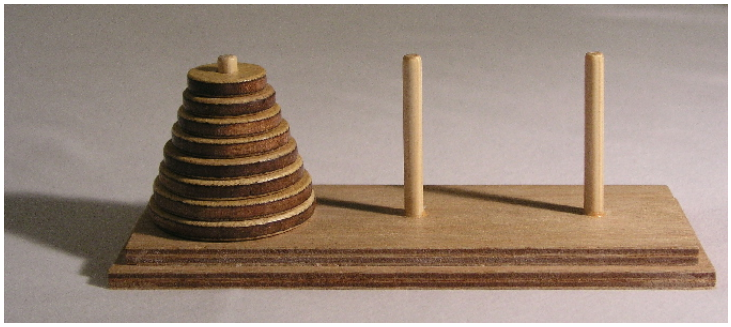

Figure 1: A model set of the Tower of Hanoi with 8 disks (source: Wikipedia)

Write a `hanoi.py` program to move the given number (from command line argument) of disks from one rod to another using recursion.

What is functional programming?

Functional programming is a separate paradigm to imperative or object orientated programming. It involves the decomposition of the problem into a set of functions that do not have an internal state. Although python supports this paradigm, it not necessarily the dominant one. Python is much more heavily influenced by imperative and object oriented paradigms.

However we can dabble in some functional programming. Typically if programming in this style it is not a particularly good idea to mix it with imperative or object orientated styles, as it leads to a confusing mess of code that conceals significant memory issues.

Our general goal here is to not evaluate any of our generators until the very end.

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.

```
def apply_function(f, *args, **kwargs):
  return f(*args, **kwargs)

apply_function(print, "hello world!")
```

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)
```

We can build our own by implementing the `__next__` method on a class. This is called by the `next()` dispatch method.

Example of an 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
```

Generators

We will introduce the `yield` keyword into python.

There is a distinct difference between generators and iterators. Iterators require the implementation of `__next__` while a generator creates an iterator, typically by calling the of `__iter__` method using the `iter` dispatch method.

A generator function wraps a regular function and passes it to the iterator object, calling `next` then calls the wrapped function and returns the current output. This is done using the `yield` keyword.

Generators will save the current state of their function call upon invoking the yield keyword. This function call is 'wrapped' and returns an object with the same interface as an iterator (it includes the `__next__` property).

```
def my_range(x):
  i = 0
  while i < x:
    yield i
    i += 1

for i in my_range(10):
  print(i)
```

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 where it yielded.

If the function returns then the iteration comes to an end.

```
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 10: C Generators

Write an even number generator in C, you will need to use a struct and dispatch methods for this.

```
int even = 0;
for (struct my_gen = g; (even = next(g));)
{
    printf("%d\n", even);
}
```

List Comprehension and Tuple Comprehension

We can further overload our `[]` and `()` operators.

A list comprehension in Python is performed using the for loop syntax and the `[]` brackets:

```
  a_list = [i * 2 for i in [1,2,3,4,5]]
```

However of much more interest to us is the ability to create a generator with much the same syntax:

```
  a_generator = (i * 2 for i in [1,2,3,4,5])
```

Here the execution is delayed until the element is required from the generator. As previously discussed, the right hand element of the for loop syntax is any iterable object, which itself could be a generator, allowing us to further delay executing or storing any objects in memory.

It is generally preferable to use a generator over a list comprehension where possible.

Question 11: 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:

```
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 $[0, n)$

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

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

```
def m_range(*args):
```

Question 12: 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:

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

Example of usage:

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

Question 13: 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 permit 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 at `0` respectively.
- Returns an iterator or a generator so it can be used in a for loop like a regular list.

```
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.

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 acts as a generator and returns an iterator object.

We can force the resolution of an iterator by wrapping list on that iterator object, it will move through each element and add them to a list.

However this leads to a large overhead in memory and should be avoided where possible.

```
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 14: 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 15: 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 16: 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.

Rather than passing a list to your filter, try passing an iterator, compare very large lists against very large iterators. What do you notice?

Hint: Try an inner lambda expression.

Partial

A very interesting and useful pattern for functional programming.

```
def partial(func, *args, **kwargs):
  def p(*w_args, **w_kwargs):
    return func(*args, *w_args, **kwargs, **w_kwargs)
  return p

f = partial(print, end='!')
f("Hello world")
```

The pattern becomes even more useful as we abstract it further

```
f = partial(partial)
g = f(print, "Hello")
h = partial(g, end='')
k = partial(h, sep=',')
g("World")
```

We can use this pattern to halt the execution of a function, while its arguments are still preserved, and combine this with other functions to modify how they act. This is incredibly powerful.

Here's an example using partial to map a map.

```
x = [[1,2,3], [4,5,6], [7,8,9]]
map(partial(map, lambda x: x * 2), x)
```

And while it is not advised you can also force it to evaluate as a list.

```
list(map(list, map(partial(map, lambda x: x * 2), x)))
```

This should give you a bit of a taste for the core concepts and conceits of functional programming.

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