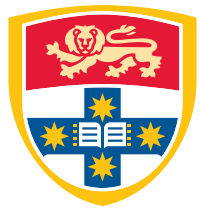
Introduction to Programming (Adv)

School of Computer Science, University of Sydney



Copyright Warning

Lecture 11: Functional programming

Generators

Generators

Producing a stream of data

Generators

Creating iterators is easy if you have the freedom to define new types and can save the state of the iterator object

What if you already had a function which produced the `__next__()` equivalent, but you didn't want to go through the trouble?

The function itself already has state built in. When you have local variables, and they possess values, that reflects the state of execution

Generators (cont.)

What is the desk check for this code?

```
1 text = "You're in the newspaper business?"
2 i = 0
3 while i < len(text):
4     ch = text[i]
5     i += 1
```

Fundamentally, each iteration produces one character that is useful as the next element in the sequence of `text`

What do we need to do to produce an iterator for the above?

Generators (cont.)

We want that output, then go off and do something with it, then come back and continue

The `yield` keyword does this

Generators (cont.)

```
1  def get_character(text):
2      i = 0
3      while i < len(text):
4          ch = text[i]
5          yield (ch)
6          i += 1
7
8  text = "You're in the newspaper business?"
9  text_generator = get_character(text)
10 while True:
11     try:
12         ch = text_generator.__next__()
13         print(ch, end='')
14     except StopIteration:
15         break
16
17 print('')
```

Generators (cont.)

That was easy!
With a generator, we also have an iterable. Meaning we can use for loops

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

Any function containing a `yield` keyword is a generator function
Generators are identified during the compilation process
Generators

1. Suspend execution of function, preserving the state of all local variables, and the position of last instruction executed
2. Return the object/value to the caller of the function
3. Upon being called again, restore the state of the function and resume execution

Simpler generators

Write a generator to return a number sequence. Each number returned is every odd number starting from the value `start`. If start is an even number, the first odd number is `start + 1`.

Example using the generator:

```
1 for i in odd_numbers(5):
2     if i > 10:
3         break
4     print(i)
```

Output:

```
5
7
9
```

Simpler generators (cont.)

```
1  def odd_numbers(start):
2      i = start
3      if i % 2 == 0:
4          i += 1
5
6      while True:
7          yield (i)
8          i += 2
```

When does it end?

it does not have to, that is up to the user of the generator

Using iterators for programming idioms

filter and map

Built in functions with iterators

There are already built in functions in Python that can act on iterable objects. These can be iterators, generator iterators, or other collection types (supported by Python)

`sum(iterable[, start])` - Sums start and the items of an iterable from left to right and returns the total.

`len()` - Return the length (the number of items) of an object.

`max(iterable, *[, key, default])` - Return the largest item in an iterable.

filter

Programming idiom find all items matching criteria `A`

Example: return only those strings that start with "EZ"

Example: return all the numbers that are in the range [0, 50)

filter (cont.)

Example: return all the numbers that are in the range [0, 50)

```
1  def get_0_to_50(numbers):
2      results = []
3      for n in numbers:
4          if n >= 0 and n < 50:
5              results.append(n)
6
7      return results
```

Example: return only those strings that start with "EZ"

```
1  def get_ez(strings):
2      results = []
3      for s in strings:
4          if len(s) < 2:
5              continue
6          if s[0] == 'E' and s[1] == 'Z':
7              results.append(s)
8
9      return results
```

filter (cont.)

`filter(function, iterable)` - Construct an iterator from those elements of iterable for which function returns true.

Example: return all the numbers that are in the range $[0, 50)$

```
1  def is_0_to_50(n):
2      if n >= 0 and n < 50:
3          return True
4      return False
5
6  nums = [0, -1, 4, 55, 25, 49, 50]
7  result = list(filter(is_0_to_50, nums))
8  print(result)
```

filter (cont.)

Example: return only those strings that start with ”EZ”

```
1  def is_ez(s):
2      if len(s) < 2:
3          return False
4      if s[0] == 'E' and s[1] == 'Z':
5          return True
6      return False
7
8  strings = ["ezasd", "EZabs", "EZ", "EzZ", "abd", ""]
9  result = list(filter(is_ez, strings))
10 print(result)
```

map

Programming idiom transform *each input element* in a way defined by function `f()` and produce a *new collection*

Example: add one to each value

Example: convert each value to a string

map (cont.)

Example: add one to each value

```
1  def add_one(numbers):
2      results = []
3      for num in numbers:
4          results.append(num + 1)
5
6      return results
7
8  nums = [0, 1, 2, 3, 4]
9  nums2 = add_one(nums)
10 print(nums2)
```

map (cont.)

Example: convert each value to a string

```
1  def convert_strings(objects):
2      results = []
3      for obj in objects:
4          results.append(str(obj))
5
6      return results
7
8  nums = [0, 1, 2, 3, 4]
9  strings = convert_strings(nums)
10 print(strings)
```

map (cont.)

`map(function, iterable, ...)` Return an iterator that applies function to every item of iterable, yielding the results.

Example: add one to each value

```
1 def add_one(n):
2     return n + 1
3
4 nums = [0, 1, 2, 3, 4]
5 nums2 = list(map(add_one, nums))
6 print(nums2)
```

Example: convert each value to a string

```
1 def convert_string(obj):
2     return str(obj)
3
4 nums = [0, 1, 2, 3, 4]
5 strings = list(map(convert_string, nums))
6 print(strings)
```

reduce

Programming idiom taking two parameters to be *accumulated* to one. When applied on a collection of n elements, it will operate a function on pairs of elements to return *a single element*.

The `sum()`, `max()` and `min()` functions are examples of reduce operations

Example: the average value. Has multiple input values, output is one number.

Standard average calculation

```
1  def average(numbers):
2      if len(numbers) < 1:
3          return 0
4      avg = 0
5      for num in numbers:
6          avg += num
7      avg = avg / len(numbers)
8      return avg
```

reduce (cont.)

Reduce style calculation

```
1  def average(numbers):
2      result = initial_value
3      for item in collection:
4          result = function(result, item)
5      return result
```

What does your `function` look like?

reduce (cont.)

`reduce(function, iterable[, initializer])` *Apply function of two arguments cumulatively to the items of iterable, from left to right, so as to reduce the iterable to a single value.*

There is not always a solution for many to one. In this case, thinking about this carefully, the average is `sum(numbers) / len(numbers)`. That is one reduce operation performed already.

Reduce style calculation with our own sum

```
1 from functools import reduce
2 def mysum(x,y):
3     return x+y
4
5 numbers = [0, 1, 2, 3, 4]
6 print(reduce(mysum, numbers)/len(numbers))
```

Functional programming

No side-effects, just functions

Functional programming

Perform all operations as functions using only input and output

Do not store any state information outside the function call (side effects)

The goal is to transform the input data to *output* using a well chosen set of functions

Many benefits: security, testing, readability, scalability (complexity)

e.g. Write a program to print the length of each element in a given list of strings

Functional programming (cont.)

e.g. Write a program to print the length of each element in a given list of strings

Procedural

Functional programming (cont.)

```
1  def get_lengths(strings):
2      results = []
3      i = 0
4      while i < len(strings):
5          results.append( len(strings[i]) )
6          i += 1
7
8      return results
9
10
11 strings = [ "I", "set", "the", "toaster", "to", "three", "-", "
       medium", "brown." ]
12 results = get_lengths(strings)
13 i = 0
14 while i < len(results):
15     print(results[i], end=' ')
16     i += 1
```

Functional programming (cont.)

e.g. Write a program to print the length of each element in a given list of strings

Object oriented

```
1  class StringCollection:
2      def __init__(self):
3          self.strings = []
4
5      def set(self, str):
6          self.strings = strs
7
8      def get_lengths(self):
9          results = []
10         i = 0
11         while i < len(strings):
12             results.append( len(strings[i]) )
13             i += 1
14
15         return results
```

Functional programming (cont.)

```
    def print_lengths(self):
        results = get_lengths(strings)
        i = 0
        while i < len(results):
            print(results[i], end=' ')
            i += 1

strc = StringCollection()
strc.set( [ "I", "set", "the", "toaster", "to", "three", "-", "
    medium", "brown." ] )
strc.print_lengths()
```

Functional programming (cont.)

e.g. Write a program to print the length of each element in a given list of strings

Functional (non-pure)

Functional programming (cont.)

```
1  def get_lengths(strings):
2      results = []
3      i = 0
4      while i < len(strings):
5          results.append( len(strings[i]) )
6          i += 1
7
8      return results
9
10 def print_list(list_in):
11     i = 0
12     while i < len(list_in):
13         print(list_in[i], end=' ')
14         i += 1
15
16 print_list( get_lengths( [ "I", "set", "the", "toaster", "to", "
       three", "-", "medium", "brown." ] ) )
```

Functional (non-pure, moving toward it)

Functional programming (cont.)

```
1 def print_list(list_in):
2     i = 0
3     while i < len(list_in):
4         print(list_in[i], end=' ')
5         i += 1
6
7 print_list( map( len, [ "I", "set", "the", "toaster", "to", "
      three", "-", "medium", "brown." ] ) )
```

Functional (pure)

```
1 print(*list(map(len, [ "I", "set", "the", "toaster", "to", "three
      ", "-", "medium", "brown." ] )))
```

Functional programming (cont.)

Functional (pure)

```
1 print(*list(map(len, [ "I", "set", "the", "toaster", "to", "three
      ", "-", "medium", "brown." ] )))
```

There are 4 functions used to solve this problem.

- Each is very simple.
- Does one thing well.
- Has input, has output.
- There is no state stored inside the functions being used
- There are no temporary variables needed to transfer the output of one function as the input to another
- The entire program is one function call and this will produce a result.

Another example

Write a program to print the lines of a file that are between 30 and 50 characters long where only the first half of that line. In the case of an odd number of characters, use the first $\frac{n}{2}$, where n is the line length. Use functional programming.

We have functions from Python to help solve this problem `print()`, `len()`, `filter()`, `map()`, `list()`

Another example (cont.)

We need to define what is a good line based on length? Focus on that problem

```
1  def good_line(line):
2      llen = len(line)
3      if llen >= 30 and llen < 50:
4          return True
5      return False
```

How do we get the first half of a line? Focus on that problem

```
1  def first_half(line):
2      llen = len(line)
3      half = llen//2 + 1
4      return line[:half]
```

We also need to traverse the file contents as strings. There are a few ways.

```
1  open("data.txt").readlines()
```

Another example (cont.)

Start from the source. Get all the data.

```
open("data.txt").readlines()
```

Reduce the source to the set of lines we are interested in (filter)

```
filter(good_line, open("data.txt").readlines())
```

Extract the first half of the line from all items (map)

```
map(first_half, filter(good_line, open("data.txt").readlines()))
```

Compose a list of the remaining data (list)

```
list(map(first_half, filter(good_line, open("data.txt").readlines
    ())))
```

print the list (print)

```
print(list(map(first_half, filter(good_line, open("data.txt").
    readlines()))))
```

A note on small code

Small can be good. Not always.

Optimising code for performance is almost always a wasted effort in early stages of software development. It is dangerous for readability and maintainability of software

Optimising code for readability is king. Almost all industry wants maintainable software products with large development teams.

Small code can be good if it really simple and comprehensible. For example `x += 1` is `x = x + 1`. `for` loop vs a `while` loop, list comprehensions can easily become obscure.

Lambda functions

lambda functions are a way to define a function without a formal declaration. It is also known as an anonymous function, it does not have a name, and is defined in the place where it is used

Example

```
1 def x_mult_y(x,y):
2     return x * y
3
4 z = x_mult_y(3, 5)
```

This is most useful with our reduce operation

```
1 from functools import reduce
2 z = reduce(lambda x,y: x*y, [3, 5])
```

Lambda functions (cont.)

Let's revisit our previous example

```
1  def good_line(line):
2      llen = len(line)
3      if llen >= 30 and llen < 50:
4          return True
5      return False
```

Can become

```
1  def good_line(line):
2      return len(line) >= 30 and len(line) < 50
```

This means we can define an anonymous function as so:

```
1  lambda line: len(line) >= 30 and len(line) < 50
```

Lambda functions (cont.)

You can go crazy with these. It will not improve your code.

```
print(list(map(first_half, filter(lambda line: len(line) >= 30
    and len(line) < 50, open("data.txt").readlines()))))
```

and also to `first_half`:

```
print(list(map(lambda line: line[:(len(line)//2 + 1)], filter(
    lambda line: len(line) >= 30 and len(line) < 50, open("data.
    txt").readlines()))))
```

Yes, this is a pure functional implementation of the program. Please don't!