

COMP1531

10.1 - Week 10 General

Iterators & Generators

- Evan Kohilas gave a more extensive talk on this topic during a CSESoc supported event a couple of weeks ago - but we will cover them lightly today

Iterators

- In Python, iterators are objects *containing* a countable number of elements
- For example, we can get an iterator for a list:

```
1 animals = ["dog", "cat", "chicken", "sheep"]  
2  
3 animal_iterator = iter(animals)
```

Iterators

- Any object with the methods `__iter__()` and `__next__()` is an iterator
- Simple example (squares)

```
1 class Squares:
2     def __init__(self):
3         self.i = 0
4
5     def __iter__(self):
6         return self
7
8     def __next__(self):
9         self.i += 1
10        return self.i*self.i
```

For loops

- Python for loops use iterators behind the scenes
- This is valid code:

```
1 squares = Squares()  
2  
3 for i in squares: # Loops forever  
4     print(i)
```

Iterator vs Iterable

- Intuitively:
 - An iterator stores the state of the iteration (i.e. where it's up to).
 - Something is iterable if it can be iterated over.
- Concretely:
 - An iterator has `__iter__()` and `__next__()` methods.
 - Iterables have `__iter__()` methods
- Thus, all iterators are iterable, but not all iterables are iterators
- For example, lists are iterable, but they are not iterators
- For loops only need to be given something *iterable*

Generators

- A different way of writing iterators
- Defined via generator functions instead of classes
- Example generator

```
1 def simple_generator():  
2     print("Hello")  
3     yield 1  
4     print("Nice to meet you")  
5     yield 2  
6     print("I am a generator")
```

Generators

- Intuitively, you can think of a generator as a suspendable computation
- Calling `next()` on a generator executes it until it reaches a `yield`, at which point it is suspended (frozen) until the subsequent call to `next()`

Generators

- More useful examples

```
1 def squares():
2     i = 0
3     while True:
4         i += 1
5         yield i*i
```

```
1 def fib():
2     a = 1
3     b = 1
4     while T
```

Libraries

- Most code re-use is through libraries.
- Software engineering can be an exercise in composing libraries to do what we want.
- This is necessary for building *useful* software.
- What's the downside?

Case study: leftpad

- A Javascript library that had many users, mostly indirect
- Owing to a disagreement, the author removed the library from NPM
- This caused thousands of Javascript-based applications and libraries to break

The entire library

```
1 module.exports = leftpad;
2 function leftpad (str, len, ch) {
3   str = String(str);
4   var i = -1;
5   if (!ch && ch !== 0) ch = ' ';
6   len = len - str.length;
7   while (++i < len) {
8     str = ch + str;
9   }
10  return str;
11 }
```

Further reading

- An analysis of the leftpad incident
 - <https://www.davidhaney.io/npm-left-pad-have-we-forgotten-how-to-program/>
- Dependency Hell
 - https://en.wikipedia.org/wiki/Dependency_hell
- An attempt fix to dependency hell
 - <https://nixos.org/nix/>