LECTURE 43

GENERATORS

MCS 275 Spring 2022
Emily Dumas

LECTURE 43: GENERATORS

Course bulletins:

- Please **complete your course evaluations**. The deadline is 11:55pm Sunday.
- Project 4 due Friday at 6pm.
- Generators demo notebook.

LOOSE END

I've converted the example program `urlreadtext.py` to a nicer version `fetch.py` that uses `argparse`.

SEQUENCES

In Python, a **sequence** is an object containing elements that can be accessed by a nonnegative integer index.

e.g. `list`, `tuple`, `str`

ITERABLES

An **iterable** is a more general concept for an object that can provide items one by one when used in a `for` loop.

Sequences can do this, but there are other examples:

iterable	**value**
file	line of text
sqlite3.Cursor*	row
dict	key

range	integer

* That's the return type of `.execute(...)` in `sqlite3`.

Unlike a sequence, an iterable may not store (or know) the next item until it is requested.

This is called laziness and can provide significant advantages.

THE IDEA

Generators are do-it-yourself lazy iterables.

THE RETURN STATEMENT

In a function, `return x` will:

- Destroy all local variables from the function (except when references to them exist in objects still in scope)
- Return execution to wherever it was when the function was called
- Replace function call with `x` for the purposes of evaluation

THE YIELD STATEMENT

When a function call is used as an iterable, the statement `yield x` will:

- **Pause** the function
- Make `x` the next value given by the iterable

The next time a value is needed, execution of the function will continue from where it left off.

COMPARISON WITH PRINT

Imagine you can write a function which will print a bunch of values (perhaps doing calculations along the way).

If you change `print(x)` to `yield x`, then you get a function that can be used as an iterable, lazily producing the same values.

GENERATOR OBJECTS

Behind the scenes, a function containing `yield` will return a **generator** object (just once), which is an iterable.

It contains the local state of the function, and to provide a value it runs the function until the next `yield`.

APPLICATIONS

- Efficient iterables when items are expensive
- Representing infinite sequences
- Retain laziness despite complex logic to determine next element (e.g. nested loops)

CONVERSION TO A SEQUENCE

The `list` and `tuple` constructors accept an iterable.

So if `g` is a generator object, `list(g)` will pull all of its items and put them in a list.

ONE-SHOT

Generator objects are "one-shot" iterables, i.e. you can only iterate over them once.

Since generator objects are usually return values of functions, it is typical to have the function call in the loop that performs iteration.

SINGLE STEPPING

The built-in function `next` will get the next value from an iterable (e.g. generator object).

It raises `StopIteration` if no more items are available.

DELEGATION

A generator can temporarily delegate to another generator, i.e. say "take values from this other generator until it is exhausted".

The syntax is

```
yield from GENERATOR
```

which is approximately equivalent to:

```
for x in GENERATOR:
    yield x
```

GENERATOR EXPRESSIONS

You can often remove the brackets from a list comprehension to get a **generator comprehension**; it behaves similarly but evaluates lazily.

```
# Create a list, then sum it
# Uses memory proportional to N
sum([ x**2 for x in range(1,N+1) ])

# Create a generator, then sum values
# it yields.  Memory usage independent
# of N.
sum( x**2 for x in range(1,N+1) )
```

This won't work in a context that needs a sequence (e.g. in `len()`, `random.choice()`, ...).

REFERENCES

- Chapter 20 of Lutz
- Chapter 4 of Beazley and Jones

REVISION HISTORY

- 2022-04-27 Initial publication
- 2022-04-29 Add link to demo notebook