Lecture 25: Coroutines

Marvin Zhang
08/03/2016

Roadmap

- Introduction
- Functions
- Data
- Mutability
- Objects
- Interpretation
- Paradigms
- Applications

Announcements

Event-Driven Programming

- Almost all programs we have seen so far involve the program running in isolation until completion
- But many practical applications involve communication between different programs or with a user
  - For example, many web applications have to wait for user input, such as mouse clicks or text input
  - We have seen one example of this: interactive interpreters wait for the user to type in code before it can execute that code and produce a result
  - This style of programming is called event-driven, because different events, such as user input, trigger different parts of our program to execute

Generators and Generator Functions

Revisiting lazy evaluation

Generator Functions

- A generator function is a function that yields values instead of returning them
- A normal function returns once, a generator function can yield multiple times
- When a generator function is called, it returns a generator that iterates over `yield` statements

```python
def range_gen(start, end):
    while start < end:
        yield start
        start += 1

>>> for i in range_gen(0, 5):
    ...    print(i)
0
1
2
3
4
```
Generators

- A generator is an iterator, created by a generator function
- Generators act as implicit, or lazy, sequences
- Values are not computed when the sequence is created, but when they are asked for
- This is the same as built-in Python `range` objects, Python iterators, and Scheme streams
- We can use implicit sequences to create infinite sequences!

```python
def naturals():
    curr = 0
    while True:
        yield curr
        curr += 1
```

```
>>> n = naturals()
>>> n
<generator object naturals at ...>
```

```
>>> next(n)
0
```

```
>>> next(n)
1
```

Generators vs Iterators

- Generator functions are often simpler and more intuitive to write than iterator classes, because:
  - We only have to write a function instead of a class
  - Yielding pauses execution of the function and automatically saves state for resuming, as opposed to returning
- Recall the iterable interface from lab: `__iter__` and `__next__`
  - `__iter__` returns an iterator, which has a `__next__` method
  - `__next__` returns the next element in our sequence
- A generator function returns a generator, which is an iterator, and the generator returns the next element by calling `__next__` on it
- So, what if we just make our `__iter__` method a generator function? This satisfies all our requirements!

Coroutines

- Generator functions can also consume values using the `yield` expression (different from the `yield` statement!)
- Generators that both produce and consume values are called coroutines, though they are still generator objects
- We can control coroutines by using the `send` and `close` methods
  - `send`, like `__next__`, resumes the coroutine, but also passes a value to it
  - Calling `__next__` is equivalent to calling `send` with `None`
- `close` stops the coroutine and raises a `GeneratorExit` exception within the coroutine

Sequence Processing

- Implicit sequences are extremely useful in programming applications that deal with continuous streams of data, e.g., news feeds, sensor measurements, or mathematical sequences
- When working with data streams, a helpful and efficient technique is to set up a pipeline for sequence processing
- One way to set up a pipeline is to have each stage of the pipeline be a coroutine!
  - Functions at the beginning of the pipeline, that only send values, are called producers
  - Coroutines in the middle, that both send and receive values, are called filters
  - Coroutines at the end of the pipeline, that only receive values, are called consumers
- The data coming through the stream is sent through this pipeline to produce the final result
Event-Driven Programming

With and without coroutines

• The paradigm of event-driven programming allows different events, such as user input, to trigger different parts of our program to execute
• Lazy evaluation, such as implicit sequences, is similar to this paradigm in that the "event" of asking for an element from the sequence triggers the computation
• However, this is not what is usually meant by "event"
• Processing continuous data streams is an example of this paradigm, where incoming data is the event
• Interactive interpreters is another example, where user input is the event
• In event-driven programming, an event loop waits for events, and handles them by dispatching them to a callback function

Interactive Interpreters (demo)

• The read-eval-print loop is an example of an event loop
• So, we can implement it using coroutines!
• This doesn't provide an advantage in this case, because the REPL is already fairly simple and elegant
  • But it is still an interesting exercise
• Let's take a look at the Calculator interpreter

Summary

• Coroutines naturally enforce modularity in our code, i.e., splitting complex functionality up into smaller pieces that are easier to write, maintain, and understand
• Modularity also allows us to easily change our program, simply by swapping in and out different pieces
• Coroutines are especially useful in building modular pipelines, where data is processed in stages
• Both generators and coroutines maintain their own state, and this is highly useful for particular applications
• Though coroutines by themselves are not a paradigm, they are useful for the paradigm of event-driven programming
• However, it is important to understand when using coroutines may just be unnecessarily complicated

Summary

• Event-driven programming is a heavily used paradigm in applications such as user interfaces and web development
• In event-driven programming, an event loop handles particular events, such as user input, and uses callback functions to process these events
• One option for implementing callback functions, which often works well, is to use coroutines
  • If the event-driven application has callback functionality that:
    • Is complex and easily made modular,
    • Naturally fits into a processing pipeline, or
    • Involves state that changes over time,
  • Then coroutines are probably the way to go