Passing Data Into Generators

What’s a Coroutine?

A **coroutine** is a computer science concept… a generalization of a function or subroutine.

Unlike regular functions, coroutines can resume execution after they "return".

And in Python, generator functions are a form of coroutine.

But there’s more to coroutines - and to generator functions - than what we’ve discussed so far.

Mental Model

Think of the flow of control "bouncing" between the generator function and the for loop.

Like two different threads. Except that only one is running at a time, and they're cooperating.

```
def gen_squares(max_root):
    root = 0
    while root < max_root:
        yield root**2
        root += 1

for square in gen_squares(5):
    print(square)
```

Notice also: data (the square number) is sent from `gen_squares()` to the for loop. But not the other direction.

The other way

It turns out:

In full coroutines, data can go **both ways**.

The coroutine (generator function) can send data to the consumer...

AND the consumer can send data **into** the coroutine.

Python supports this!

Receiving and Printing

```python
def receive_and_print():
    print("Starting...")
    while True:
        payload = (yield)
        print("RECEIVED: " + payload)

receiver = receive_and_print()
# Have to "prime" the coroutine with next()
next(receiver)

receiver.send("hey")
receiver.send("yay")
```

Notice:

- The "`payload = (yield)`" line
- The generator object has a `send()` method

Yielding

```python
def receive_and_print():
    print("Starting...")
    while True:
        payload = (yield)
        print("RECEIVED: " + payload)
```

```
>>> receiver = receive_and_print()
>>> next(receiver)
Starting...
>>>
```

Sending

```
def receive_and_print():
    print("Starting...")
    while True:
        payload = (yield)
        print("RECEIVED: " + payload)
```

```
>>> receiver.send("hey")
RECEIVED: hey
>>> receiver.send("yay")
RECEIVED: yay
```

Practice: receive_and_print()

Create a file called `receiveandprint.py`. Type in the following:

```
def receive_and_print():
    print("Starting...")
    while True:
        payload = (yield)
        print("RECEIVED: " + payload)

receiver = receive_and_print()
next(receiver)

receiver.send("Python")
receiver.send("rocks")
```

Run the program. Output should be:

```
Starting...
RECEIVED: Python
RECEIVED: rocks
```

Closing

Generator objects also have a method called `close()`.

This causes the generator object (coroutine) to immediately exit... so no more values can be sent.

```
>>> receiver = receive_and_print()
>>> next(receiver)
Starting...
>>>
>>> receiver.close()
>>> receiver.send("one more")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

But you can also make it do special "clean-up" on closing.

GeneratorExit

```python
# Version 2.
def receive_and_print2():
    print("Starting...")
    try:
        while True:
            payload = (yield)
            print("RECEIVED: " + payload)
    except GeneratorExit:
        print("CLOSING")
        # "raise" is optional in this case, because the next line will
        # exit the function anyway. But for other gen functions, you
        # might need it.
        raise
```

GeneratorExit

```
>>> receiver = receive_and_print2()
>>> next(receiver)
Starting...
>>> receiver.send("hey")
RECEIVED: hey
>>> receiver.send("yay")
RECEIVED: yay
>>> receiver.close()
CLOSING
```

Using finally

A common pattern: instead of catching `GeneratorExit`, use a "finally" block for cleanup.

```
def receive_and_print2():
    print("Starting...")
    try:
        while True:
            payload = (yield)
            print("RECEIVED: " + payload)
    finally:
        print("CLOSING")
        # The coroutine will naturally exit,
```

Only works when the flow of control will exit after the finally block.

Bi-Directional Data Flow

Coroutines are meant to be bi-directional in their data flow.

Meaning: A coroutine can return a value to the caller.

And: The caller can also pass a value back into the coroutine.

Also: The coroutine can raise an exception, that the caller must handle.

But:

If a coroutine is bi-directional… doesn’t that mean the caller can raise an exception in the other direction?

Raising an exception **into** the coroutine?

Throwing in exceptions

```python
# Version 3.
def receive_and_print3():
    print("Starting...")
    while True:
        try:
            payload = (yield)
        except ValueError:
            payload = "[INVALID]"
        print("RECEIVED: " + payload)
```

```
>>> receiver = receive_and_print3()
>>> next(receiver)
Starting...
>>>
>>> receiver.throw(ValueError)
RECEIVED: [INVALID]
```

Terminology

In computer science, a **generator** implies a kind of function that scalably *produces* values.

A **coroutine** is a kind of function that can *accept* values, as well as produce them.

In Python, you *implement* both "generators" (produce values) and "coroutines" (consume values) by writing a **generator function**.

Strictly speaking, "coroutine" is more general than "generator".

Generators Vs. Coroutines

Both generators (producing values) and coroutines (accepting them via `.send()`) are implemented with the `yield` keyword.

It’s best to think of these as filling separate roles: data sinks vs. data sources.

(Though it’s possible for one generator function to do both at the same time, as you’ll see later.)

Reminder: house_records()

Remember this generator function - it produces a stream of dictionaries, with keys like "address", "square_feet", and "price_usd":

```python
def house_records(path):
    with open(path) as lines:
        record = {}
        for line in lines:
            if line == '\n':
                yield record
                record = {}
                continue
            key, value = line.rstrip('\n').split(': ', 1)
            record[key] = value
        yield record
```

Let’s build a coroutine to scalably handle what it produces.

Processing pipelines

```python
def make_db_writer(username, password):
    conn = dbconnect(username, password)
    try:
        while True:
            record = (yield)
            conn.sql(
                "INSERT INTO house_sales (location, price) VALUES (?, ?)",
                record["address"], record["price_usd"])
    finally:
        conn.commit()
        conn.close()

# db writer sink coroutine
db_writer = make_db_writer(USERNAME, PASSWORD)
next(db_writer)

for record in house_records("housedata.txt"):
    db_writer.send(record)
db_writer.close()
```

Write CSV

Imagine we also want to write a CSV output file of records, but adding a numeric index - like this:

```
index,address,square_feet,price_usd
1,1423 99th Ave,1705,340210
2,24257 Pueblo Dr,2305,170210
3,127 Cochran,2068,320500
...
```

How could we write a new coroutine for that?

Write CSV

```python
import csv
def make_csv_writer(dest_path):
    with open(dest_path, 'w') as dest:
        writer = csv.DictWriter(dest, fieldnames=[
            "index", "address", "square_feet", "price_usd"])
        writer.writeheader()
        index = 1
        while True:
            row = (yield)
            row["index"] = index
            writer.writerow(row)
            index += 1

csv_writer = make_csv_writer(CSV_RECORD_FILE)
next(csv_writer)

for record in house_records("housedata.txt"):
    csv_writer.send(record)
```

No need to `.close()` this one. But it won’t hurt anything if we do.

Fanning Out Writing

One great thing about coroutines: you can easily fan out to multiple data sinks.

Here’s one that can chain two writers like `db_writer` and `csv_writer` together:

```
def record_house_records(writer1, writer2):
    try:
        while True:
            record = (yield)
            writer1.send(record)
            writer2.send(record)
    except GeneratorExit:
        writer1.close()
        writer2.close()
```

Chaining

```
# Create one coroutine that uses all writers
writers = record_house_records(db_writer, csv_writer)
next(writers)

for record in house_records("housedata.txt"):
    writers.send(record)
```

This inserts all rows in the database, AND creates the CSV file!

Generalizing

Of course, we want to generalize this, to take as many or few writers as we want.

We can do it this way:

```
def chain(*writers):
    try:
        while True:
            record = (yield)
            for writer in writers:
                writer.send(record)
    except GeneratorExit:
        for writer in writers:
            writer.close()
```

Then just create it with:

```
writers = chain(db_writer, csv_writer)
```

Yielding & Receiving

It’s possible for a generator object to both yield and receive values.

Simply replace `(yield)` with `(yield VALUE)`. Then `send()` returns that value.

```python
def make_parrot():
    message = "Polly wants a cracker"
    while True:
        squawk = message.upper() + "!"
        message = (yield squawk)
```

Parroting

```
>>> parrot = make_parrot()
>>>
>>> # The first, default message is returned when it's primed.
... next(parrot)
'POLLY WANTS A CRACKER!'
>>> # Additional sends return the yielded squawk.
... parrot.send("Hello")
'HELLO!'
>>> parrot.send("Stop copying me")
'STOP COPYING ME!'
```

You can… but should you?

There are use cases for generator objects that simultaneously receive and produce values.

But often, the complexity seems not to be worth it.

My recomendation:

When you write a generator function, decide whether it will yield (produce) values; or receive them (via `send()`). Pick one or the other.

Only do both when the coding situation truly requires it.

Sub-generators

Sometimes you'll have a generator internally using another generator.

```python
def evens(limit):
    num = 0
    while num <= limit:
        yield num
        num += 2

def evens_under_10():
    for num in evens(10):
        yield num
```

For a pure producer, this works fine.

But it won't forward calls to `send()`, `throw()` or `close()`.

Delegating with "yield from"

You can instead do what's called **delegating to a subgenerator**.

Do this with the `yield from` keyword:

```
def evens_under_10():
    yield from evens(10)
```

For producing values, it's similar to this:

```
def evens_under_10():
    for num in evens(10):
        yield num
```

But `yield from` does more.

Forwarding send()

```
def prefix_printer(prefix):
    while True:
        message = (yield)
        print(prefix + message)

def warner():
    yield from prefix_printer('WARNING: ')

warn = warner()
next(warn)

warn.send('Disk at 95% capacity')
warn.send('Network timeout')
```

The `yield from` "keyword"

This of "`yield from`" as a single keyword, different from `yield`.

(I almost think it should have been "`yieldfrom`", to make the distinction clearer.)