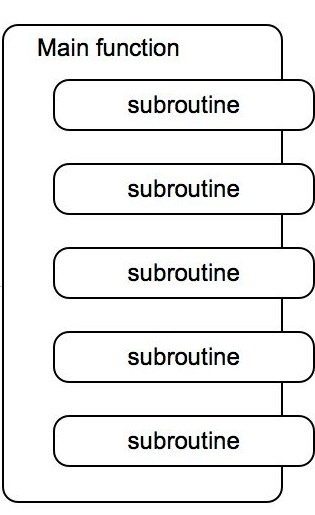
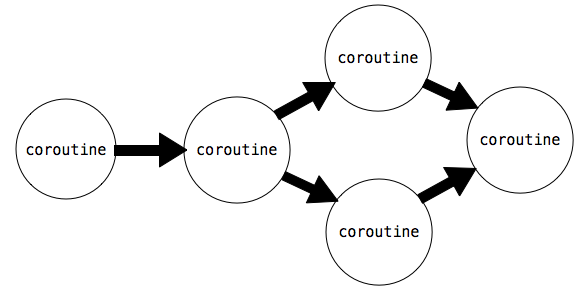
**Coroutines**





## Python Coroutines

The generator functions, uses yield to return values. Python generator functions can also consume values using a(yield) statement. In addition two new methods on generator objects, send() and close(), create a framework for objects that consume and produce values. Generator functions that define these objects are coroutines.

Coroutines consume values using a (yield) statement as follows:

value = (yield)

With this syntax, execution pauses at this statement until the object's send method is invoked with an argument:

coroutine.send(data)

Then, execution resumes, with value being assigned to the value of data. To signal the end of a computation, we shut down a coroutine using the close()method. This raises a GeneratorExit exception inside the coroutine, which we can catch with a try/except clause.

The example below illustrates these concepts. It is a coroutine that prints strings that match a provided pattern.

>>> def match(pattern):

print('Looking for ' + pattern)

try:

while True:

s = (yield)

if pattern in s:

print(s)

except GeneratorExit:

print("=== Done ===")

We initialize it with a pattern, and call \_\_next\_\_() to start execution:

>>> m = match("Jabberwock")

>>> m.\_\_next\_\_()

Looking for Jabberwock

The call to \_\_next\_\_() causes the body of the function to be executed, so the line "Looking for jabberwock" gets printed out. Execution continues until the statement line = (yield) is encountered. Then, execution pauses, and waits for a value to be sent to m. We can send values to it using send.

>>> m.send("the Jabberwock with eyes of flame")

the Jabberwock with eyes of flame

>>> m.send("came whiffling through the tulgey wood")

>>> m.send("and burbled as it came")

>>> m.close()

=== Done ===

When we call m.send with a value, evaluation resumes inside the coroutine m at the statement line = (yield), where the sent value is assigned to the variable line. Evaluation continues inside m, printing out the line if it matches, going through the loop until it encounters line = (yield) again. Then, evaluation pauses inside m and resumes where m.send was called.

We can chain functions that send() and functions that yield together achieve complex behaviors. For example, the function below splits a string named text into words and sends each word to another coroutine.

>>> def read(text, next\_coroutine):

for line in text.split():

next\_coroutine.send(line)

next\_coroutine.close()

Each word is sent to the coroutine bound to next\_coroutine, causing next\_coroutine to start executing, and this function to pause and wait. It waits until next\_coroutine pauses, at which point the function resumes by sending the next word or completing.

If we chain this function together with match defined above, we can create a program that prints out only the words that match a particular word.

>>> text = 'Commending spending is offending to people pending lending!'

>>> matcher = match('ending')

>>> matcher.\_\_next\_\_()

Looking for ending

>>> read(text, matcher)

Commending

spending

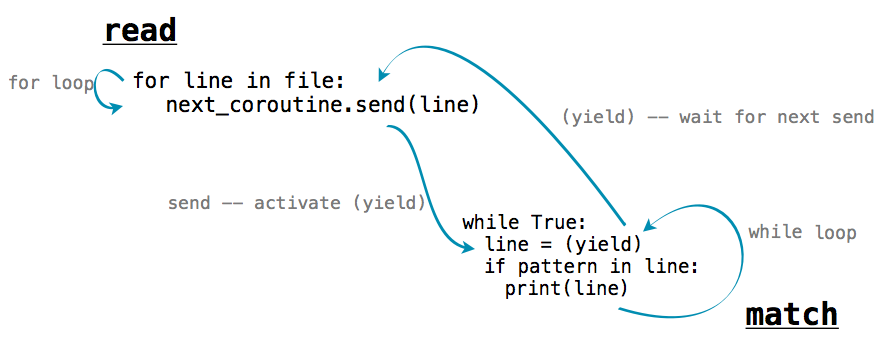
offending

pending

lending!

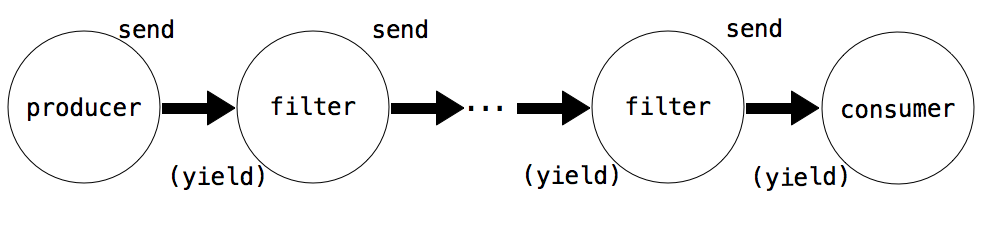
=== Done ===

The read function sends each word to the coroutine matcher, which prints out any input that matches its pattern. Within the matcher coroutine, the line s = (yield) waits for each sent word, and it transfers control back to read when it is reached.



## 5.3.2   Produce, Filter, and Consume

Coroutines can have different roles depending on how they use yield and send():



* A **Producer** creates items in a series and uses send(), but not (yield)
* A **Filter** uses (yield) to consume items and send() to send result to a next step.
* A **Consumer** uses (yield) to consume items, but does not send.

The function read above is an example of a producer. It does not use (yield), but uses send to produce data items. The function match is an example of a consumer. It does not send anything, but consumes data with (yield).We can break up match into a filter and a consumer. The filter would be a coroutine that only sends on strings that match its pattern.

>>> def match\_filter(pattern, next\_coroutine):

print('Looking for ' + pattern)

try:

while True:

s = (yield)

if pattern in s:

next\_coroutine.send(s)

except GeneratorExit:

next\_coroutine.close()

And the consumer would be a function that printed out lines sent to it.

>>> def print\_consumer():

print('Preparing to print')

try:

while True:

line = (yield)

print(line)

except GeneratorExit:

print("=== Done ===")

When a filter or consumer is constructed, its \_\_next\_\_ method must be invoked to start its execution.

>>> printer = print\_consumer()

>>> printer.\_\_next\_\_()

Preparing to print

>>> matcher = match\_filter('pend', printer)

>>> matcher.\_\_next\_\_()

Looking for pend

>>> read(text, matcher)

spending

pending

=== Done ===

Even though the name filter implies removing items, filters can transform items as well. The function below is an example of a filter that transforms items. It consumes strings and sends along a dictionary of the number of times different letters occur in the string.

>>> def count\_letters(next\_coroutine):

try:

while True:

s = (yield)

counts = {letter:s.count(letter) for letter in set(s)}

next\_coroutine.send(counts)

except GeneratorExit as e:

next\_coroutine.close()

We can use it to count the most frequently-used letters in text using a consumer that adds up dictionaries and finds the most frequent key.

>>> def sum\_dictionaries():

total = {}

try:

while True:

counts = (yield)

for letter, count in counts.items():

total[letter] = count + total.get(letter, 0)

except GeneratorExit:

max\_letter = max(total.items(), key=lambda t: t[1])[0]

print("Most frequent letter: " + max\_letter)

To run this pipeline on a file, we must first read the lines of a file one-by-one. Then, we send the results through count\_letters and finally tosum\_dictionaries. We can re-use the read coroutine to read the lines of a file.

>>> s = sum\_dictionaries()

>>> s.\_\_next\_\_()

>>> c = count\_letters(s)

>>> c.\_\_next\_\_()

>>> read(text, c)

Most frequent letter: n

## 5.3.3   Multitasking

A producer or filter does not have to be restricted to just one next step. It can have multiple coroutines downstream of it, and send() data to all of them. For example, here is a version of read that sends the words in a string to multiple next steps.

>>> def read\_to\_many(text, coroutines):

for word in text.split():

for coroutine in coroutines:

coroutine.send(word)

for coroutine in coroutines:

coroutine.close()

We can use it to examine the same text for multiple words:

>>> m = match("mend")

>>> m.\_\_next\_\_()

Looking for mend

>>> p = match("pe")

>>> p.\_\_next\_\_()

Looking for pe

>>> read\_to\_many(text, [m, p])

Commending

spending

people

pending

=== Done ===

=== Done ===

First, read\_to\_many calls send(word) on m. The coroutine, which is waiting at text = (yield) runs through its loop, prints out a match if found, and resumes waiting for the next send. Execution then returns to read\_to\_many, which proceeds to send the same line to p. Thus, the words of text are printed in order.