Generator Patterns

Generator Funcs and Files

Here's a little program. Focus on `matchinglines()`:

```
# findpattern.py
import sys

def matchinglines(pattern, path):
    with open(path) as handle:
        for line in handle:
            if pattern in line:
                yield line.rstrip('\n')

pattern, path = sys.argv[1], sys.argv[2]
for line in matchinglines(pattern, path):
    print(line)
```

Using matchinglines

`log.txt`:

```
WARNING: Disk usage exceeding 85%
DEBUG: User 'tinytim' upgraded to Pro version
INFO: Sent email campaign, completed normally
WARNING: Almost out of beer
```

`matchinglines("WARNING:", "log.txt")` yields:

```
WARNING: Disk usage exceeding 85%
WARNING: Almost out of beer
```

It does this on ANY size log file, with a small memory footprint.

Records

The log file contains **records**, one per line:

```
WARNING: Disk usage exceeding 85%
DEBUG: User 'tinytim' upgraded to Pro version
...
```

Suppose your application needs that data in dict form:

```
{"level": "WARNING", "message": "Disk usage exceeding 85%"}
{"level": "DEBUG", "message": "User 'tinytim' upgraded to Pro version"}
```

How do you scalably transform log lines to dictionaries?

Transformation

You want to **transform** the data, from one form to another:

```
def parse_log_records(lines):
    for line in lines:
        level, message = line.split(": ", 1)
        yield {"level": level, "message": message}
```

Chain it together with `matchinglines`:

```
log_lines = matchinglines("WARNING:", "logfile.txt")
for record in parse_log_records(log_lines):
    print(record)
```

```
{'level': 'WARNING', 'message': 'Disk usage exceeding 85%'}
{'level': 'WARNING', 'message': 'Almost out of beer'}
```

Building Blocks

But it can also be used on its own.

```
with open("logfile.txt") as handle:
    for record in parse_log_records(handle):
        print(record)
```

```
{'level': 'WARNING', 'message': 'Disk usage exceeding 85%'}
{'level': 'DEBUG', 'message': 'User "tinytim" upgraded to Pro version'}
{'level': 'INFO', 'message': 'Sent email campaign, completed normally'}
{'level': 'WARNING', 'message': 'Almost out of beer'}
```

`matchinglines` and `parse_log_records` are building blocks, which can be used to build different data processing streams.

Scalable Composability

Call this *scalable composability*.

It goes beyond designing composable functions and types.

Ask yourself how you can make the components scalable, **and** whatever is assembled out of them scalable too.

Generator Functions!

Interfaces

How you might integrate these in a class:

```
class Logs:
    def __init__(self, logfile_path):
        self.logfile_path = logfile_path
    def records(self):
        with open(self.logfile_path) as log_lines:
            for record in parse_log_records(log_lines):
                yield record
def warnings(self):
    log_lines = matchinglines("WARNING:", self.logfile_path)
    for record in parse_log_records(log_lines):
        yield record
```

Transforming Adapter

You can think of `parse_log_records` as an **adapter**... transforming records in one form (lines) to a more useful one (dictionaries).

Well-structured programs have many such boundaries of transformation.

Generator functions are an excellent device for creating them.

Comprehensions (our next topic) are surprisingly relevant.

Record Mapping

You can think of generator functions as mapping one stream of records to another stream.

With `parse_log_records`, one input record maps to one output record:

- **Input record**: one *line*
- **Output record**: one *dict*

What happens when the mapping **isn't** one-to-one?

- Several input records are consumed to produce one output record. Or...
- One input record creates several output records

Fan Out: Word Parsing

Imagine a text file containing lines in a poem:

```
all night our room was outer-walled with rain
drops fell and flattened on the tin roof
and rang like little disks of metal
...
```

Let's create a generator function, `words_in_text`, producing the words one at a time.

words_in_text

First approach:

```
def words_in_text(path):
    with open(path) as handle:
        for line in handle:
            line = line.rstrip('\n')
            for word in line.split():
                yield word
```

There is a potential bottleneck in here. What is it? How can we do better?

Looooong lines

```
def words_in_text(path):
    BUFFER_SIZE = 2**20
    def read(): return handle.read(BUFFER_SIZE)
    def normalize(chunk): return chunk.lower().rstrip(',!.\n')
    with open(path) as handle:
        buffer = read()
        start, end = 0, -1
        while True:
            for match in re.finditer(r'[ \t\n]+', buffer):
                end = match.start()
                yield normalize(buffer[start:end])
                start = match.end()
            new_buffer = read()
            if new_buffer == '':
                break # end of file
            buffer = buffer[end+1:] + new_buffer
            start, end = 0, -1
word = normalize(buffer[start:])
if word != '':
    yield word
```

Fan In: House Sale Data

`housedata.txt`: One key-value pair per line, with records separated by blank lines.

```
address: 1423 99th Ave
square_feet: 1705
price_usd: 340210

address: 24257 Pueblo Dr
square_feet: 2305
price_usd: 170210

address: 127 Cochran
square_feet: 2068
price_usd: 320500
```

Generating House Records

We want a generator function called `house_records`, which will read this data in, and give us a stream of dictionaries:

```
>>> houses = house_records("housedata.txt")
>>> house = next(houses)
>>> type(house)
<class 'dict'>
>>> list(house.keys())
['address', 'square_feet', 'price_usd']
>>> house['address']
'1423 99th Ave'
>>> house = next(houses)
>>> house['address']
'24257 Pueblo Dr'
```

Reading House Records

```
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
```

Lab: Generator Adapters

Lab file: `generators/adapter.py`

- In labs/py3 for 3.x; labs/py2 for 2.7
- When you are done, give a thumbs up...
- ... and then do `generators/adapter_extra.py`

The methods of str are detailed here:
https://docs.python.org/3/library/stdtypes.html#string-methods