A practical introduction to functional programming

Mary Rose Cook

Many functional programming articles teach abstract functional techniques. That is, composition, pipelining, higher order functions. This one is different. It shows examples of imperative, unfunctional code that people write every day and translates these examples to a functional style.

The first section of the article takes short, data transforming loops and translates them into functional maps and reduces. The second section takes longer loops, breaks them up into units and makes each unit functional. The third section takes a loop that is a long series of successive data transformations and decomposes it into a functional pipeline.

The examples are in Python, because many people find Python easy to read. A number of the examples eschew pythonicity in order to demonstrate functional techniques common to many languages: map, reduce, pipeline. All of the examples are in Python 2.

A guide rope

When people talk about functional programming, they mention a dizzying number of "functional" characteristics. They mention immutable data[1], first class functions[2] and tail call optimisation[3]. These are language features that aid functional programming. They mention mapping, reducing, pipelining, recursing, currying[4] and the use of higher order functions. These are programming techniques used to write functional code. They mention parallelization[5], lazy evaluation[6] and determinism[7]. These are advantageous properties of functional programs.

Ignore all that. Functional code is characterised by one thing: the absence of side effects. It doesn't rely on data outside the current function, and it doesn't change data that exists outside the current function. Every other "functional" thing can be derived from this property. Use it as a guide rope as you learn.

This is an unfunctional function:

```
a = 0
```

```
def increment():

    global a

    a += 1
```

This is a functional function:

```
def increment(a):

    return a + 1
```

Don’t iterate over lists. Use map and reduce.

Map

Map takes a function and a collection of items. It makes a new, empty collection, runs the function on each item in the original collection and inserts each return value into the new collection. It returns the new collection.

This is a simple map that takes a list of names and returns a list of the lengths of those names:

```
name_lengths = map(len, ["Mary", "Isla", "Sam"])



print name_lengths

# => [4, 4, 3]
```

This is a map that squares every number in the passed collection:

```
squares = map(lambda x: x * x, [0, 1, 2, 3, 4])



print squares
```

```
# => [0, 1, 4, 9, 16]
```

This map doesn't take a named function. It takes an anonymous, inlined function defined with `lambda`. The parameters of the lambda are defined to the left of the colon. The function body is defined to the right of the colon. The result of running the function body is (implicitly) returned.

The unfunctional code below takes a list of real names and replaces them with randomly assigned code names.

```
import random



names = ['Mary', 'Isla', 'Sam']

code_names = ['Mr. Pink', 'Mr. Orange', 'Mr. Blonde']



for i in range(len(names)):

    names[i] = random.choice(code_names)



print names

# => ['Mr. Blonde', 'Mr. Blonde', 'Mr. Blonde']
```

(As you can see, this algorithm can potentially assign the same secret code name to multiple secret agents. Hopefully, this won't be a source of confusion during the secret mission.)

This can be rewritten as a map:

```
import random



names = ['Mary', 'Isla', 'Sam']
```

```
secret_names = map(lambda x: random.choice(['Mr. Pink',
                                            'Mr. Orange',
                                            'Mr. Blonde']),
                   names)
```

Exercise 1. Try rewriting the code below as a map. It takes a list of real names and replaces them with code names produced using a more robust strategy.

```
names = ['Mary', 'Isla', 'Sam']


for i in range(len(names)):
    names[i] = hash(names[i])


print names
# => [6306819796133686941, 8135353348168144921, -1228887169324443034]
```

(Hopefully, the secret agents will have good memories and won't forget each other's secret code names during the secret mission.)

My solution:

```
names = ['Mary', 'Isla', 'Sam']


secret_names = map(hash, names)
```

Reduce

Reduce takes a function and a collection of items. It returns a value that is created by combining the items.

This is a simple reduce. It returns the sum of all the items in the collection.

```
sum = reduce(lambda a, x: a + x, [0, 1, 2, 3, 4])



print sum

# => 10
```

`x` is the current item being iterated over. `a` is the accumulator. It is the value returned by the execution of the lambda on the previous item. `reduce()` walks through the items. For each one, it runs the lambda on the current `a` and `x` and returns the result as the `a` of the next iteration.

What is `a` in the first iteration? There is no previous iteration result for it to pass along. `reduce()` uses the first item in the collection for `a` in the first iteration and starts iterating at the second item. That is, the first `x` is the second item.

This code counts how often the word `'Sam'` appears in a list of strings:

```
sentences = ['Mary read a story to Sam and Isla.',

             'Isla cuddled Sam.',

             'Sam chortled.']



sam_count = 0

for sentence in sentences:

    sam_count += sentence.count('Sam')



print sam_count

# => 3
```

This is the same code written as a reduce:

```
sentences = ['Mary read a story to Sam and Isla.',
             'Isla cuddled Sam.',
             'Sam chortled.']

sam_count = reduce(lambda a, x: a + x.count('Sam'),
                   sentences,
                   0)
```

How does this code come up with its initial `a`? The starting point for the number of incidences of `'Sam'` cannot be `'Mary read a story to Sam and Isla.'` The initial accumulator is specified with the third argument to `reduce()`. This allows the use of a value of a different type from the items in the collection.

Why are map and reduce better?

First, they are often one-liners.

Second, the important parts of the iteration - the collection, the operation and the return value - are always in the same places in every map and reduce.

Third, the code in a loop may affect variables defined before it or code that runs after it. By convention, maps and reduces are functional.

Fourth, map and reduce are elemental operations. Every time a person reads a `for` loop, they have to work through the logic line by line. There are few structural regularities they can use to create a scaffolding on which to hang their understanding of the code. In contrast, map and reduce are at once building blocks that can be combined into complex algorithms, and elements that the code reader can instantly understand and abstract in their mind. “Ah, this code is transforming each item in this collection. It’s throwing some of the transformations away. It’s combining the remainder into a single output.”

Fifth, map and reduce have many friends that provide useful, tweaked versions of their basic behaviour. For example: `filter`, `all`, `any` and `find`.

Exercise 2. Try rewriting the code below using map, reduce and filter. Filter takes a function and a collection. It returns a collection of every item for which the function returned `True`.

```
people = [{'name': 'Mary', 'height': 160},
          {'name': 'Isla', 'height': 80},
          {'name': 'Sam'}]

height_total = 0
height_count = 0
for person in people:
    if 'height' in person:
        height_total += person['height']
        height_count += 1

if height_count > 0:
    average_height = height_total / height_count

    print average_height
    # => 120
```

If this seems tricky, try not thinking about the operations on the data. Think of the states the data will go through, from the list of people dictionaries to the average height. Don't try and bundle multiple transformations together. Put each on a separate line and assign the result to a descriptively-named variable. Once the code works, condense it.

My solution:

```
people = [{'name': 'Mary', 'height': 160},
          {'name': 'Isla', 'height': 80},
          {'name': 'Sam'}]

heights = map(lambda x: x['height'],
              filter(lambda x: 'height' in x, people))

if len(heights) > 0:
    from operator import add
    average_height = reduce(add, heights) / len(heights)
```

Write declaratively, not imperatively

The program below runs a race between three cars. At each time step, each car may move forwards or it may stall. At each time step, the program prints out the paths of the cars so far. After five time steps, the race is over.

This is some sample output:

```
-
--
--

--
--
---
```

```
---

--

---



----

---

----



----

----

-----
```

This is the program:

```
from random import random



time = 5

car_positions = [1, 1, 1]



while time:

    # decrease time

    time -= 1



    print ''
```

```
    for i in range(len(car_positions)):
        # move car
        if random() > 0.3:
            car_positions[i] += 1

        # draw car
        print '-' * car_positions[i]
```

The code is written imperatively. A functional version would be declarative. It would describe what to do, rather than how to do it.

Use functions

A program can be made more declarative by bundling pieces of the code into functions.

```
from random import random


def move_cars():
    for i, _ in enumerate(car_positions):
        if random() > 0.3:
            car_positions[i] += 1


def draw_car(car_position):
    print '-' * car_position


def run_step_of_race():
```

```
    global time
    time -= 1
    move_cars()


def draw():
    print ''
    for car_position in car_positions:
        draw_car(car_position)


time = 5
car_positions = [1, 1, 1]


while time:
    run_step_of_race()
    draw()
```

To understand this program, the reader just reads the main loop. “If there is time left, run a step of the race and draw. Check the time again.” If the reader wants to understand more about what it means to run a step of the race, or draw, they can read the code in those functions.

There are no comments any more. The code describes itself.

Splitting code into functions is a great, low brain power way to make code more readable.

This technique uses functions, but it uses them as sub-routines. They parcel up code. The code is not functional in the sense of the guide rope. The functions in the code use state that was not passed as arguments. They affect the code around them by changing external variables, rather than by returning values. To check what a function really does, the reader must read each line carefully. If

they find an external variable, they must find its origin. They must see what other functions change that variable.

Remove state

This is a functional version of the car race code:

```
from random import random



def move_cars(car_positions):

    return map(lambda x: x + 1 if random() > 0.3 else x,

               car_positions)



def output_car(car_position):

    return '-' * car_position



def run_step_of_race(state):

    return {'time': state['time'] - 1,

            'car_positions': move_cars(state['car_positions'])}



def draw(state):

    print ''

    print '\n'.join(map(output_car, state['car_positions']))



def race(state):

    draw(state)

    if state['time']:
```

```
        race(run_step_of_race(state))


race({'time': 5,
      'car_positions': [1, 1, 1]})
```

The code is still split into functions, but the functions are functional. There are three signs of this. First, there are no longer any shared variables. `time` and `car_positions` get passed straight into `race()`. Second, functions take parameters. Third, no variables are instantiated inside functions. All data changes are done with return values. `race()` recurses[3] with the result of `run_step_of_race()`. Each time a step generates a new state, it is passed immediately into the next step.

Now, here are two functions, `zero()` and `one()`:

```
def zero(s):
    if s[0] == "0":
        return s[1:]


def one(s):
    if s[0] == "1":
        return s[1:]
```

`zero()` takes a string, `s`. If the first character is `'0'`, it returns the rest of the string. If it is not, it returns `None`, the default return value of Python functions. `one()` does the same, but for a first character of `'1'`.

Imagine a function called `rule_sequence()`. It takes a string and a list of rule functions of the form of `zero()` and `one()`. It calls the first rule on the string. Unless `None` is returned, it takes the return value and calls the second rule on it. Unless `None` is returned, it takes the return value and calls the third rule on it. And so forth. If any rule returns `None`, `rule_sequence()` stops and returns `None`. Otherwise, it returns the return value of the final rule.

This is some sample input and output:

```
print rule_sequence('0101', [zero, one, zero])

# => 1



print rule_sequence('0101', [zero, zero])

# => None
```

This is the imperative version of `rule_sequence()`:

```
def rule_sequence(s, rules):

    for rule in rules:

        s = rule(s)

        if s == None:

            break



    return s
```

Exercise 3. The code above uses a loop to do its work. Make it more declarative by rewriting it as a recursion.

My solution:

```
def rule_sequence(s, rules):

    if s == None or not rules:

        return s

    else:
```

```
        return rule_sequence(rules[0](s), rules[1:])
```

Use pipelines

In the previous section, some imperative loops were rewritten as recursions that called out to auxiliary functions. In this section, a different type of imperative loop will be rewritten using a technique called a pipeline.

The loop below performs transformations on dictionaries that hold the name, incorrect country of origin and active status of some bands.

```
bands = [{'name': 'sunset rubdown', 'country': 'UK', 'active': False},

         {'name': 'women', 'country': 'Germany', 'active': False},

         {'name': 'a silver mt. zion', 'country': 'Spain', 'active': True}]



def format_bands(bands):

    for band in bands:

        band['country'] = 'Canada'

        band['name'] = band['name'].replace('.', '')

        band['name'] = band['name'].title()



format_bands(bands)



print bands

# => [{'name': 'Sunset Rubdown', 'active': False, 'country': 'Canada'},

#     {'name': 'Women', 'active': False, 'country': 'Canada' },

#     {'name': 'A Silver Mt Zion', 'active': True, 'country': 'Canada'}]
```

Worries are stirred by the name of the function. “format” is very vague. Upon closer inspection of the code, these worries begin to claw. Three things happen in the same loop. The `'country'` key gets set to `'Canada'`. Punctuation is removed from the band name. The band name gets capitalized. It is hard to tell what the code is intended to do and hard to tell if it does what it appears to do. The code is hard to reuse, hard to test and hard to parallelize.

Compare it with this:

```
print pipeline_each(bands, [set_canada_as_country,
                            strip_punctuation_from_name,
                            capitalize_names])
```

This code is easy to understand. It gives the impression that the auxiliary functions are functional because they seem to be chained together. The output from the previous one comprises the input to the next. If they are functional, they are easy to verify. They are also easy to reuse, easy to test and easy to parallelize.

The job of `pipeline_each()` is to pass the bands, one at a time, to a transformation function, like `set_canada_as_country()`. After the function has been applied to all the bands, `pipeline_each()` bundles up the transformed bands. Then, it passes each one to the next function.

Let’s look at the transformation functions.

```
def assoc(_d, key, value):
    from copy import deepcopy
    d = deepcopy(_d)
    d[key] = value
    return d


def set_canada_as_country(band):
    return assoc(band, 'country', "Canada")
```

```
def strip_punctuation_from_name(band):
    return assoc(band, 'name', band['name'].replace('.', ''))


def capitalize_names(band):
    return assoc(band, 'name', band['name'].title())
```

Each one associates a key on a band with a new value. There is no easy way to do this without mutating the original band. `assoc()` solves this problem by using `deepcopy()` to produce a copy of the passed dictionary. Each transformation function makes its modification to the copy and returns that copy.

Everything seems fine. Band dictionary originals are protected from mutation when a key is associated with a new value. But there are two other potential mutations in the code above. In `strip_punctuation_from_name()`, the unpunctuated name is generated by calling `replace()` on the original name. In `capitalize_names()`, the capitalized name is generated by calling `title()` on the original name. If `replace()` and `title()` are not functional, `strip_punctuation_from_name()` and `capitalize_names()` are not functional.

Fortunately, `replace()` and `title()` do not mutate the strings they operate on. This is because strings are immutable in Python. When, for example, `replace()` operates on a band name string, the original band name is copied and `replace()` is called on the copy. Phew.

This contrast between the mutability of strings and dictionaries in Python illustrates the appeal of languages like Clojure. The programmer need never think about whether they are mutating data. They aren't.

Exercise 4. Try and write the `pipeline_each` function. Think about the order of operations. The bands in the array are passed, one band at a time, to the first transformation function. The bands in the resulting array are passed, one band at a time, to the second transformation function. And so forth.

My solution:

```
def pipeline_each(data, fns):
```

```
    return reduce(lambda a, x: map(x, a),
                  fns,
                  data)
```

All three transformation functions boil down to making a change to a particular field on the passed band. `call()` can be used to abstract that. It takes a function to apply and the key of the value to apply it to.

```
set_canada_as_country = call(lambda x: 'Canada', 'country')
strip_punctuation_from_name = call(lambda x: x.replace('.', ''), 'name')
capitalize_names = call(str.title, 'name')

print pipeline_each(bands, [set_canada_as_country,
                            strip_punctuation_from_name,
                            capitalize_names])
```

Or, if we are willing to sacrifice readability for conciseness, just:

```
print pipeline_each(bands, [call(lambda x: 'Canada', 'country'),
                            call(lambda x: x.replace('.', ''), 'name'),
                            call(str.title, 'name')])
```

The code for `call()`:

```
def assoc(_d, key, value):
    from copy import deepcopy
    d = deepcopy(_d)
```

```
    d[key] = value
    return d


def call(fn, key):
    def apply_fn(record):
        return assoc(record, key, fn(record.get(key)))
    return apply_fn
```

There is a lot going on here. Let's take it piece by piece.

One. `call()` is a higher order function. A higher order function takes a function as an argument, or returns a function. Or, like `call()`, it does both.

Two. `apply_fn()` looks very similar to the three transformation functions. It takes a record (a band). It looks up the value at `record[key]`. It calls `fn` on that value. It assigns the result back to a copy of the record. It returns the copy.

Three. `call()` does not do any actual work. `apply_fn()`, when called, will do the work. In the example of using `pipeline_each()` above, one instance of `apply_fn()` will set `'country'` to `'Canada'` on a passed band. Another instance will capitalize the name of a passed band.

Four. When an `apply_fn()` instance is run, `fn` and `key` will not be in scope. They are neither arguments to `apply_fn()`, nor locals inside it. But they will still be accessible. When a function is defined, it saves references to the variables it closes over: those that were defined in a scope outside the function and that are used inside the function. When the function is run and its code references a variable, Python looks up the variable in the locals and in the arguments. If it doesn't find it there, it looks in the saved references to closed over variables. This is where it will find `fn` and `key`.

Five. There is no mention of bands in the `call()` code. That is because `call()` could be used to generate pipeline functions for any program, regardless of topic. Functional programming is partly about building up a library of generic, reusable, composable functions.

Good job. Closures, higher order functions and variable scope all covered in the space of a few paragraphs. Have a nice glass of lemonade.

There is one more piece of band processing to do. That is to remove everything but the name and country. `extract_name_and_country()` can pull that information out:

```
def extract_name_and_country(band):
    plucked_band = {}
    plucked_band['name'] = band['name']
    plucked_band['country'] = band['country']
    return plucked_band


print pipeline_each(bands, [call(lambda x: 'Canada', 'country'),
                            call(lambda x: x.replace('.', ''), 'name'),
                            call(str.title, 'name'),
                            extract_name_and_country])


# => [{'name': 'Sunset Rubdown', 'country': 'Canada'},
#     {'name': 'Women', 'country': 'Canada'},
#     {'name': 'A Silver Mt Zion', 'country': 'Canada'}]
```

`extract_name_and_country()` could have been written as a generic function called `pluck()`. `pluck()` would be used like this:

```
print pipeline_each(bands, [call(lambda x: 'Canada', 'country'),
                            call(lambda x: x.replace('.', ''), 'name'),
                            call(str.title, 'name'),
                            pluck(['name', 'country'])])
```

Exercise 5. `pluck()` takes a list of keys to extract from each record. Try and write it. It will need to be a higher order function.

My solution:

```
def pluck(keys):

    def pluck_fn(record):

        return reduce(lambda a, x: assoc(a, x, record[x]),

                      keys,

                      {})

    return pluck_fn
```

What now?

Functional code co-exists very well with code written in other styles. The transformations in this article can be applied to any code base in any language. Try applying them to your own code.

Think of Mary, Isla and Sam. Turn iterations of lists into maps and reduces.

Think of the race. Break code into functions. Make those functions functional. Turn a loop that repeats a process into a recursion.

Think of the bands. Turn a sequence of operations into a pipeline.

[1] An immutable piece of data is one that cannot be changed. Some languages, like Clojure, make all values immutable by default. Any “mutating” operations copy the value, change it and pass back the changed copy. This eliminates bugs that arise from a programmer’s incomplete model of the possible states their program may enter.

[2] Languages that support first class functions allow functions to be treated like any other value. This means they can be created, passed to functions, returned from functions and stored inside data structures.

[3] Tail call optimisation is a programming language feature. Each time a function recurses, a new stack frame is created. A stack frame is used to store the arguments and local values for the current function invocation. If a function recurses a large number of times, it is possible for the interpreter or compiler to run out of memory. Languages with tail call optimisation reuse the same stack frame for their entire sequence of recursive calls. Languages like Python that do

not have tail call optimisation generally limit the number of times a function may recurse to some number in the thousands. In the case of the `race()` function, there are only five time steps, so it is safe.

[4] Currying means decomposing a function that takes multiple arguments into a function that takes the first argument and returns a function that takes the next argument, and so forth for all the arguments.

[5] Parallelization means running the same code concurrently without synchronization. These concurrent processes are often run on multiple processors.

[6] Lazy evaluation is a compiler technique that avoids running code until the result is needed.

[7] A process is deterministic if repetitions yield the same result every time.