CIS 1100

Functional Programming in Python

Python
Fall 2024
University of Pennsylvania

Review:

We covered three general purpose higher-order functions:

- `filter`
- `map`
- `reduce`

and a new language feature: `lambda`

Today we are just going to do a bunch of practice with it and apply it to more complicated scenarios

Why?

Why are we talking about Higher Order Functions (HOF)?

It turns out that a LOT of problems we want to solve in computer science can reduce down to one of the three funnctions we have shown

- `filter`
- `map` (sometimes called `transform`)
- `reduce` (sometimes called `fold`, `accumulate`, `aggregate` or other terms)

These are sort of "fundamental" patterns in computer science, showing up in many programming languages.

If you want to take more CIS courses (e.g. CIS 1200) then this is a core topic.

`filter`

`filter` is a higher order function that takes in a function and sequence and returns a new sequence containing only those elements for which the provided function evaluates to `True`.

`filter(f, seq)` is equivalent to:

```
[elem for elem in seq if f(elem)]
```

or

```
res = []
for elem in seq:
    if f(elem):
        res.append(elem)
```

`map`

`map` is a higher order function that takes in a function and sequence and returns a new sequence containing elements of the input sequence after having `f` applied to them.

`map(f, seq)` is equivalent to:

```
[f(elem) for elem in seq]
```

or

```
res = []
for elem in seq:
    res.append(f(elem))
```

`reduce`

Aggregation is done using a slightly trickier HOF called `reduce` imported from `functools`.

`reduce` is a function that takes in an *accumulator function* and a sequence and repeatedly accumulates elements from the sequence using the accumulator function.

`reduce(f, seq)` is roughly equivalent to:

```
result = seq[0]
for elem in seq[1:]:
    result = f(result, elem)
```

Lambdas

Lambdas (or anonymous functions) are functions that are defined without names.

```
lambda <parameter_list> : expression
```

would become

```
def no_name(<parameter_list>):
    return expression
```

Practice! What do each of these evaluate to?

(S7)

```
def perfect_name(name):
    return len(name) == 5 and 'a' in name

names = ["travis", "Harry", "Jessica", "adi", "Sukya", "Molly", "CedriC", "Jared"]
result = list(filter(perfect_name, names))
```

(S8)

```
names = ["travis", "Harry", "Jessica", "adi", "Sukya", "Molly", "CedriC", "Jared"]
result = list(map(lambda name : name[0] + name[-1], names))
```

(S9)

```
music = ["7eleven", "Cinco De Mayo Shit Show", "Vomitspit", "Lefty", "All My Friends"]
music = list(filter(lambda title : len(title.split()) == 1, music))
music = reduce(lambda a, b: a += b[0], music)
```

Practice! Identifying the pattern

As said before, many problems can boil down to a combination of filter/map/reduce.

For these problems what functions do you think would be needed?

(**Select all that apply for each question**, A for filter, B for map, C for reduce)

- (M1) convert a list of strings/characters into a single string e.g. ["h", "i", "!"] -> "hi!"
- (M2) get a list of all the capital letters in a string
- (M3) given a list of numbers, get another list of numbers that has the square (square of x is x^2)
- (M4) find the maximum value in a list without using `max()`
- (M5) given a list of strings, get a single string that is made up of the first character of each string. (e.g. ["Hello!" , "I, "am", "tired"] -> "HIat")

Applying Higher Order Functions to past problems

Remember the `caesar` homework?

We wrote a function called `string_to_symbol_list()` that took a string and returned a list but all characters were converted to symbols. `ord(charater) - 65`

What does this sound like?

Try implementing it with a higher order function (L11):

```
def string_to_symbol_list(string):
    # TODO
```

Applying HOF to Caesar

```
def shift(symbol, n):
    return (symbol + n) % 26 if 0 <= symbol <= 25 else symbol

# assume `n` is the amount we want to shift each symbol by
def encrypt(to_encode, n):
    symbols = string_to_symbol_list(to_encode)
    # TODO: put something here
    return symbol_list_to_string(symbols)
```

Which of these would work for encrypt? (M6)

- (A) `symbols = list(map(shift, symbols))`
- (B) `symbols = list(map(lambda char : shift(char, n), symbols))`
- (C) `symbols = list(map(shift(n), symbols))`
- (D) `symbols = list(reduce(shift, symbols, []))`

(Bonus: When we implement `symbol_list_to_string`, which HOF will we need?)

Practice Code writing!

Given a list of characters, take only the ones which are singular lowercase letters and mash them up into character*3 in a string. e.g. [a,A,B,c,b,travis] becomes "aaacccbbb".

(C12)

Applied to more complex structures

Remember the structure we used in the check-in?

We asked you to implement the `def get_test_names(autograder_results):` which got a list of all tests in that structure.

Which higher order functions would apply here?

```
{ "score": 44.0,
  "execution_time": 136, # optional, seconds
  "output": "Text relevant to the entire submission", # optional
  "output_format": "simple_format",
  "test_output_format": "text",
  "test_name_format": "text",
  "visibility": "after_due_date",
  "stdout_visibility": "visible",
  "tests":
    [
        {
            "score": 1.0,
            "max_score": 2.0,
            "status": "failed",
            "name": "Name of the first test here",
            "name_format": "text",
            "number": "1.1",
            "output": "test output",
            "output_format": "text",
            "visibility": "visible", # Optional visibility setting
            "extra_data": {} # Optional extra data to be stored
        },
        # another test case
        {
            "score": 2.0,
            "max_score": 2.0,
            "status": "passed",
            "name": "Name of the second test here",
            "name_format": "text",
            "number": "1.2",
            "output": "THIS IS THE OUTPUT OF THIS TEST",
            "output_format": "text",
            "visibility": "visible", # Optional visibility setting
            "extra_data": {} # Optional extra data to be stored
        },

        # and more test cases...
    ],
}
```

Applied to more complex structures

Remember the structure we used in the check-in?

We asked you to implement the `def get_test_names(autograder_results):` which got a list of all tests in that structure.

If we used HOF we could do:

```
def get_test_names(autograder_results):
    return list(map(lambda test : test['name'], autograder_results['tests']))
```

```
{ "score": 44.0,
  "execution_time": 136, # optional, seconds
  "output": "Text relevant to the entire submission", # optional
  "output_format": "simple_format",
  "test_output_format": "text",
  "test_name_format": "text",
  "visibility": "after_due_date",
  "stdout_visibility": "visible",
  "tests":
    [
        {
            "score": 1.0,
            "max_score": 2.0,
            "status": "failed",
            "name": "Name of the first test here",
            "name_format": "text",
            "number": "1.1",
            "output": "test output",
            "output_format": "text",
            "visibility": "visible", # Optional visibility setting
            "extra_data": {} # Optional extra data to be stored
        },
        # another test case
        {
            "score": 2.0,
            "max_score": 2.0,
            "status": "passed",
            "name": "Name of the second test here",
            "name_format": "text",
            "number": "1.2",
            "output": "THIS IS THE OUTPUT OF THIS TEST",
            "output_format": "text",
            "visibility": "visible", # Optional visibility setting
            "extra_data": {} # Optional extra data to be stored
        },

        # and more test cases...
    ],
}
```

Applied to more complex structures

Remember the structure we used in the check-in?

We asked you to implement the `def get_failing_test_names(autograder_results):` which got a list of all failing tests in that structure.
(e.g. the status was "failed")

How would we do it with HOF?
(C14)

```
{ "score": 44.0,
  "execution_time": 136, # optional, seconds
  "output": "Text relevant to the entire submission", # optional
  "output_format": "simple_format",
  "test_output_format": "text",
  "test_name_format": "text",
  "visibility": "after_due_date",
  "stdout_visibility": "visible",
  "tests":
    [
        {
            "score": 1.0,
            "max_score": 2.0,
            "status": "failed",
            "name": "Name of the first test here",
            "name_format": "text",
            "number": "1.1",
            "output": "test output",
            "output_format": "text",
            "visibility": "visible", # Optional visibility setting
            "extra_data": {} # Optional extra data to be stored
        },
        # another test case
        {
            "score": 2.0,
            "max_score": 2.0,
            "status": "passed",
            "name": "Name of the second test here",
            "name_format": "text",
            "number": "1.2",
            "output": "THIS IS THE OUTPUT OF THIS TEST",
            "output_format": "text",
            "visibility": "visible", # Optional visibility setting
            "extra_data": {} # Optional extra data to be stored
        },

        # and more test cases...
    ],
}
```

Next time!

Recursion :)