Fundamental Concepts of Programming Languages

Functional Programming Fundamentals
Lecture 13

conf. dr. ing. Ciprian-Bogdan Chirila

University Politehnica Timisoara
Department of Computing and Information Technology

January 10, 2023

FCPL - 13 - Functional Programming in Practice

FCPL - 13 - Functional Programming in Practice

Avoiding flow control

- a block of code contains:
  - outside loops like for or while
  - assignment of state variables within loops
  - modification of data structures
  - branch statements if, elif, else, try, except, finally
- it seems natural and easy
- problems with side effects due to state variables and mutable data structures
  - a mutable object **can** be changed after its creation
  - an immutable object **cannot** be changed after its creation

Avoiding flow control

- the problem
  - it difficult to reason accurately about what state data is in at a given point in a program
- the solution
  - is not to focus on the data construction
  - but on describing what the data collection consists of
- imperative flow control is about the "how" rather than the "what"
- to focus on "what" by refactoring the code
- to pus the data construction in a more isolated place

Encapsulation

```
# configure the data to start with
collection = get_initial_state()
state_var = None
for datum in data_set:
 if condition(state_var):
  state_var = calculate_from(datum)
  new = modify(datum, state_var)
  collection.add_to(new)
 else:
  new = modify_differently(datum)
  collection.add_to(new)

# Now actually work with the data
for thing in collection:
 process(thing)
```

Encapsulation

```
# tuck away construction of data
def make_collection(data_set):
 collection = get_initial_state()
 state_var = None
 for datum in data_set:
  if condition(state_var):
   state_var = calculate_from(datum, state_var)
   new = modify(datum, state_var)
   collection.add_to(new)
  else:
   new = modify_differently(datum)
   collection.add_to(new)
 return collection

# Now actually work with the data
for thing in make_collection(data_set):
 process(thing)
```

Encapsulation

- there is no program logic change
- we shifted from **how** do we construct the collection
- to **what** does `make_collection()` create

FCPL - 13 - Functional Programming in Practice

Comprehensions

- are a way to make the code more compact
- they shift the focus from **how** to **what**
- are expressions that use the same keywords as loop and conditional blocks
- inverts their order to focus on the data rather than on the procedure
- changing the form of the expression makes a large difference in how we reason about the code

Comprehensions

```
collection = list()
for datum in data_set:
 if condition(datum):
  collection.append(datum)
 else:
  new = modify(datum)
  collection.append(new)
---
collection = [d if condition(d) else modify(d)
 for d in data_set]
```

Comprehensions

- we saved a few characters and lines :)
- we did a mental shifting by thinking what a collection is
- we avoided to think about the state of collection in the loop
- in Python there are several types of comprehensions
  - generator comprehensions;
  - set comprehensions;
  - dict comprehensions.
- as caveat nesting comprehensions may stop clarifying and start obscuring
- the solution is to refactor into functions

FCPL - 13 - Functional Programming in Practice

Generator comprehensions

- have almost the same syntax as list comprehensions
- there are no square brackets around them, but parentheses
- they are also **lazy**
- they represent a description of how to get the data
- but it is not realized until one explicitly asks for it
  - by calling .next() on the object
  - by looping over it
- saves memory for large sequences
- defers computation until is actually needed

Generator comprehensions

```
log_lines = (line for line in read_line(huge_log_file)
 if complex_condition(line))
---
# the imperative version
def get_log_lines(log_file):
 line = read_line(log_file)
 while True:
  try:
   if complex_condition(line):
    yield line
   line = read_line(log_file)
  except StopIteration:
   raise

log_lines = get_log_lines(huge_log_file)
```

Generator comprehensions

```
# another imperative version
class GetLogLines(object):

 def __init__(self, log_file):
  self.log_file = log_file
  self.line = None

 def __iter__(self):
  return self

 def __next__(self):
  if self.line is None:
   self.line = read_line(log_file)
  while not complex_condition(self.line):
   self.line = read_line(self.log_file)
  return self.line

log_lines = GetLogLines(huge_log_file)
```

FCPL - 13 - Functional Programming in Practice

Dictionaries and sets comprehensions

```
>>> {i:chr(65+i) for i in range(6)}
{0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E', 5: 'F'}

>>> {chr(65+i) for i in range(6)}
{'A', 'B', 'C', 'D', 'E', 'F'}
```

FCPL - 13 - Functional Programming in Practice

Recursion

- functional programming is about expressing flow control using recursion instead of loops
- thus, we can avoid altering the state of any variable within the algorithm
- recursion can be iteration having just another name
  - it is in the style of Lisp
  - it is not in the style of Python (slow at recursion and has limited stack depth sys.setrecursionlimit(5000) default is 1000)
- recursion can be used in solving problems by partitioning into smaller problems
- Python lacks *tail call elimination* feature

Example of recursion being iteration

```
def running_sum(numbers, start=0):
 if len(numbers) == 0:
  print()
  return
 total = numbers[0] + start
 print(total, end=" ")
 running_sum(numbers[1:], total)
```

Example of recursion being iteration

- this approach is not recommended
- the iteration which modifies the `total` state variable is more readable
- it is likely to call the function on sequences larger than 1000

Recursion less trivial example

```
def factorialR(N):
 "Recursive factorial function"
 assert isinstance(N, int) and N >= 1
 return 1 if N <= 1 else N * factorialR(N-1)

def factorialI(N):
 "Iterative factorial function"
 assert isinstance(N, int) and N >= 1
 product = 1
 while N >= 1:
  product *= N
  N -= 1
 return product
```

High order functions

```
from functools import reduce
from operator import mul

def factorialHOF(n):
 return reduce(mul, range(1, n+1), 1)
```

Quicksort example

```
def quicksort(lst):
 "Quicksort over a list-like sequence"
 if len(lst) == 0:
  return lst
 pivot = lst[0]
 pivots = [x for x in lst if x == pivot]
 small = quicksort([x for x in lst if x < pivot])
 large = quicksort([x for x in lst if x > pivot])
 return small + pivots + large
```

FCPL - 13 - Functional Programming in Practice

Eliminating loops

- we to try to eliminate all loops from a Python program
- this practice is not always desirable because it affects readability
- it is simple to apply it in a systematic manner
- if we find a function call inside a loop we can use the high order function `map()`
- there is no repeated binding of the iteration variable

Statement and map based loop

```
for e in it: # statement-based loop
 func(e)
---
map(func, it) # map()-based "loop"
```

Statement and map based loop

```
# let f1, f2, f3 (etc) be functions that perform actions
# an execution utility function
do_it = lambda f, *args: f(*args)

# map()-based action sequence
map(do_it, [f1, f2, f3])
```

Using map

```
>>> hello = lambda first, last: print("Hello", first, last)
>>> bye = lambda first, last: print("Bye", first, last)
>>> _ = list(map(do_it, [hello, bye], ['David','Jane'], ['Mertz','Doe']))
Hello David Mertz
Bye Jane Doe
---
>>> do_all_funcs = lambda fns, *args: [list(map(fn, *args)) for fn in fns]
>>> _ = do_all_funcs([hello, bye],['David','Jane'], ['Mertz','Doe'])
Hello David Mertz
Hello Jane Doe
Bye David Mertz
Bye Jane Doe
```

Eliminating loops

```
# statement-based while loop
while <cond>:
 <pre-suite>
 if <break_condition>:
  break
 else:
  <suite>
---
# FP-style recursive while loop
def while_block():
 <pre-suite>
 if <break_condition>:
  return 1
 else:
  <suite>
 return 0

while_FP = lambda: (<cond> and while_block()) or while_FP()
while_FP()
```

FCPL - 13 - Functional Programming in Practice

Streams

- streams are wrappers around a data source
- allow us to operate with the data source
- allows bulk processing convenient and fast
- do not store data
- are not a data structures
- do not modify the underlying data source
- `java.util.stream` present from Java 8

Stream creation

```
private static Employee[] arrayOfEmps =
{
 new Employee(1, "Jeff Bezos", 100000.0),
 new Employee(2, "Bill Gates", 200000.0),
 new Employee(3, "Mark Zuckerberg", 300000.0)
};
Stream.of(arrayOfEmps);
---
private static List<Employee> empList = Arrays.asList(arrayOfEmps);
empList.stream();
---
// streaming from individual objects
Stream.of(arrayOfEmps[0], arrayOfEmps[1], arrayOfEmps[2]);
```

Stream builder

```
Stream.Builder<Employee> empStreamBuilder = Stream.builder();

empStreamBuilder.accept(arrayOfEmps[0]);
empStreamBuilder.accept(arrayOfEmps[1]);
empStreamBuilder.accept(arrayOfEmps[2]);

Stream<Employee> empStream = empStreamBuilder.build();
```

forEach operation

```
@Test
public void whenIncrementSalaryForEachEmployee_thenApplyNewSalary()
{
 empList.stream().forEach(e -> e.salaryIncrement(10.0));

 assertThat(empList, contains(
  hasProperty("salary", equalTo(110000.0)),
  hasProperty("salary", equalTo(220000.0)),
  hasProperty("salary", equalTo(330000.0))
 ));
}
```

map operation

```
@Test
public void whenMapIdToEmployees_thenGetEmployeeStream()
{
 Integer[] empIds = { 1, 2, 3 };

 List<Employee> employees = Stream.of(empIds)
  .map(employeeRepository::findById)
  .collect(Collectors.toList());

 assertEquals(employees.size(), empIds.length);
}
```

map operation

- produces a new stream after applying a function to each element of the original stream
- the new stream could be of different type
- in the example we converted a stream of Integers into a stream of Employees
- each Integer is passed to the function `employeeRepository::findById()`
- it returns the Employee object
- thus, it forms an Employee stream

collect operation

```
@Test
public void whenCollectStreamToList_thenGetList()
{
 List<Employee> employees = empList.stream().collect(Collectors.toList());
 assertEquals(empList, employees);
}
```

collect operation

- is the way to get the elements out of the stream once we are done with the processing
- performs mutable fold operations on data elements held in the Stream instance
- fold operation means repacking elements to some data structures and applying additional logic

filter operation

```
@Test
public void whenFilterEmployees_thenGetFilteredStream()
{
 Integer[] empIds = { 1, 2, 3, 4 };

 List<Employee> employees = Stream.of(empIds)
  .map(employeeRepository::findById)
  .filter(e -> e != null)
  .filter(e -> e.getSalary() > 200000)
  .collect(Collectors.toList());

 assertEquals(Arrays.asList(arrayOfEmps[2]), employees);
}
```

filter operation

- produces new stream of elements that passed a given test
- the test is specified by a predicate
- in the example:
    - we filtered out the *null* references for invalid employee ids
    - we filetred out the employees having salaries under a certain threshold

findFirst operation

```
@Test
public void whenFindFirst_thenGetFirstEmployeeInStream()
{
 Integer[] empIds = { 1, 2, 3, 4 };

 Employee employee = Stream.of(empIds)
  .map(employeeRepository::findById)
  .filter(e -> e != null)
  .filter(e -> e.getSalary() > 100000)
  .findFirst()
  .orElse(null);

 assertEquals(employee.getSalary(), new Double(200000));
}
```

findFirst operation

- return an Optional innstance for the first entry in the stream
- the Optional instance may be null
- in the example:
  - we return the employee with the salary greater than a threshold
  - if no employee exists then null is returned

toArray operation

```
@Test
public void whenStreamToArray_thenGetArray()
{
 Employee[] employees = empList.stream().toArray(Employee[]::new);
 assertThat(empList.toArray(), equalTo(employees));
}
```

toArray operation

- we saw the example with the collection of elements
- we also can get an array out of the stream by using `toArray()` method
- in the example:
  - the Emplyee[]::new creates an empty array of Employee
  - it is then filled with elements from the stream

flatMap operation

```
@Test
public void whenFlatMapEmployeeNames_thenGetNameStream()
{
 List<List<String>> namesNested = Arrays.asList(
  Arrays.asList("Jeff", "Bezos"),
  Arrays.asList("Bill", "Gates"),
  Arrays.asList("Mark", "Zuckerberg"));

 List<String> namesFlatStream = namesNested.stream()
  .flatMap(Collection::stream)
  .collect(Collectors.toList());

 assertEquals(namesFlatStream.size(), namesNested.size() * 2);
}
```

flatMap operation

- a stream can hold complex data structures like Stream<List<String>>
- flatMap() helps to flatten the data structure to simplify further operations
- in the example:
  - we converted the Stream¡List¡String¿¿ to a simple Stream¡String¿

peek operation

```
@Test
public void whenIncrementSalaryUsingPeek_thenApplyNewSalary()
{
 Employee[] arrayOfEmps =
 {
  new Employee(1, "Jeff Bezos", 100000.0),
  new Employee(2, "Bill Gates", 200000.0),
  new Employee(3, "Mark Zuckerberg", 300000.0)
 };

 List<Employee> empList = Arrays.asList(arrayOfEmps);

 empList.stream()
 .peek(e -> e.salaryIncrement(10.0))
 .peek(System.out::println)
 .collect(Collectors.toList());

 assertThat(empList, contains(
 hasProperty("salary", equalTo(110000.0)),
 hasProperty("salary", equalTo(220000.0)),
 hasProperty("salary", equalTo(330000.0))
 ));
```

peek operation

- `forEach` is a terminal operation
- sometimes we need to perform multiple operations on each element before any terminal operation
- `peek()` is usefull in such situations
- in the example:
  - the first `peek()` is used to increment the salary of each employee
  - the second `peek()` is used to print the employees
  - finally `collect()` is used as terminal operation

FCPL - 13 - Functional Programming in Practice

Bibliography

1. Horia Ciocarlie - The programming language universe, second edition, Timisoara, 2013.
2. Carlo Ghezzi, Mehdi Jarayeri - Programming Languages, John Wiley, 1987.
3. Ellis Horrowitz - Fundamentals of programming languages, Computer Science Press, 1984.
4. Donald Knuth - The art of computer programming, 2002.
5. David Merz - Functional programming in Python, 2015.
6. Eugen Paraschiv - A Guide to Java Streams in Java 8: In-Depth Tutorial With Examples, 2022.