Programming Fundamentals

Functional Programming with Collections

João Correia Lopes

INESC TEC, FEUP

4 december 2018

GOALS

By the end of this class, the student should be able to:

- Describe advanced collection concepts using *Lists of Tuples*
- Simplify some common list-processing patterns using *List Comprehensions*
- Simplify some list processing using *Sequence Processing Functions*: `map()`, `filter()`
- Clarify code using *lambda forms*

BIBLIOGRAPHY

- Steven F. Lott, *Building Skills in Python — A Programmer's Introduction to Python*, FreeTechBooks, 2010 (Chapter 21) [PDF]

TIPS

- There's no slides: we use a script, illustrations and code in the class. Note that this PDF is NOT a replacement for **studying the bibliography** listed in the *class plan*
- "Students are responsible for anything that transpires during a class—therefore **if you're not in a class**, you should get notes from someone else (not the instructor)"—David Mayer
- The best thing to do is to **read carefully** and **understand** the documentation published in the Content wiki (or else **ask** in the recitation class)
- We will be using **Moodle** as the primary means of communication

CONTENTS

1 FUNCTIONAL PROGRAMMING W/ COLLECTIONS
- Lists of Tuples
- List Comprehensions
- Sequence Processing Functions: `map()`, `filter()`
- Advanced List Sorting
- The Lambda

CODE, TEST & PLAY

- Have a look at the code in GitHub:
  https://github.com/fpro-admin/lectures/
- Test before you submit at FPROtest:
  http://fpro.fe.up.pt/test/
- Pay a visit to the playground at FPROplay:
  http://fpro.fe.up.pt/play/

FUNCTIONAL PROGRAMMING

- "Programming in a functional language consists in building definitions and using the computer to evaluate expressions."[1]
- The primary role of the programmer is to construct a function to solve a give problem.
- This function, which may involve a number of subsidiary functions, is expressed in notation that obeys normal mathematical principles.
- The primary role of the computer is to act as an evaluator or calculator: its job is to evaluate expressions and print results.

[1] Bird & Wadler, Introduction to Functional Programming, Prentice-Hall, 1988

ADVANCED COLLECTION CONCEPTS

- *Lists of Tuples* — describe the relatively common Python data structure built from a list of tuples
- *List Comprehensions* — powerful list construction method used to simplify some common list-processing patterns
- `map()`, `filter()` — functions that can simplify some list processing and provide features that overlap with list comprehensions
- *lambda forms* — aren't essential for Python programming, but they're handy for clarifying a piece of code in some cases

LISTS OF TUPLES

- The list of tuple structure is remarkably useful
- One common situation is processing list of simple coordinate pairs for 2-dimensional or 3-dimensional geometries
- As an example of using a red, green, blue **tuple**, we may have a list of individual colors that looks like:

```
    colorScheme = [ (0,0,0), (0x20,0x30,0x20), (0x10,0xff,0xff) ]
```

Working with Lists of Tuples

- In dictionaries, the `dict.items()` method provides the dictionary keys and values as a `list` of 2-tuples
- The `zip()` built-in function *interleaves* two or more sequences to create a `list` of tuples
- A interesting form of the `for` statement is one that exploits multiple assignment to work with a list of tuples

```
for c,f in [("red",18), ("black",18), ("green",2)]:
    print("{0} occurs {1}".format(c, f/38.0))
```

- The `for` statement uses a form of multiple assignment to split up each tuple into a fixed number of variables

⇒ https://github.com/fpro-admin/lectures/blob/master/19/for.py

LIST DISPLAYS

- For constructing a list, a set or a dictionary, Python provides special syntax called "displays"[2]
- The most common list *display* is the simple literal value:

```
[ expression < , ... > ]
```

- For example:

```
fruit = ["Apples", "Peaches", "Pears", "Bananas"]
```

- But Python has a second kind of list *display*, based on a list comprehension

[2] The Python Language Reference

LIST COMPREHENSIONS

- A list comprehension is an expression that combines a function, a `for` statement, and an optional `if` statement
- This allows a simple, clear expression of the processing that will build up an iterable sequence
- The most important thing about a list comprehension is that it is an iterable that applies a calculation to another iterable
- A list display can use a list comprehension iterable to create a new list

```
even = [2*x for x in range(18)]
```

LIST COMPREHENSION SEMANTICS

- When we write a list comprehension, we will provide an iterable, a variable and an expression
- Python will process the iterator as if it was a for-loop, iterating through a sequence of values
- It evaluates the expression, once for each iteration of the for-loop
- The resulting values can be collected into a fresh, new list, or used anywhere an iterator is used

```python
string = "Hello 12345 World"
for n in [int(x) for x in string if x.isdigit()]:
    print(n*n)
```

List Comprehension Syntax

- A list comprehension is — technically — a complex expression
- It's often used in list displays, but can be used in a variety of places where an iterator is expected

```
expr <for-clause>
```

- The `expr` is any expression
- It can be a simple constant, or any other expression (including a nested list comprehension)
- The `for-clause` mirrors the `for` statement

```
for variable in sequence
```

COMPREHENSION IN A LIST DISPLAY

- For example:

```
even = [2*x for x in range(18)]
hardways = [(x,x) for x in (2,3,4,5)]
samples = [random.random() for x in range(10)]
```

- A list display that uses a list comprehension behaves like the following loop:

```
r = []
for variable in sequence:
    r.append(expr)
```

⇒ https://github.com/fpro-admin/lectures/blob/master/19/for_comp.py

COMPREHENSIONS OUTSIDE LIST DISPLAYS

- We can use the iterable list comprehension in other contexts that expect an iterator

```
square = sum((2*a+1) for a in range(10))

column_1 = tuple(3*b+1 for b in range(12))

# create a generator object that will iterate over 100 values
rolls = ((random.randint(1,6), random.randint(1,6)) for u in range(100))

hardways = any(d1==d2 for d1,d2 in rolls)
```

⇒ https://github.com/fpro-admin/lectures/blob/master/19/out_comp.py

The if Clause

- A list comprehension can also have an **if-clause**

```
expr <for-clause> <if-clause>
```

- Here is an example of a complex list comprehension in a list display

```
hardways = [(x,x) for x in range(1,7) if x+x not in (2, 12)]
```

- This more complex list comprehension behaves like the following loop:

```
r= []
for variable in sequence :
   if filter:
      r.append( expr )
```

Another example

```
>>> [(x, 2*x+1) for x in range(10) if x % 3 == 0]

[(0, 1), (3, 7), (6, 13), (9, 19)]
```

- This works as follows:
  1. The for-clause iterates through the 10 values given by `range(10)`, assigning each value to the local variable `x`
  2. The if-clause evaluates the filter function, `x % 3 == 0`. If it is `False`, the value is skipped; if it is `True`, the expression, at `(x, 2*x+1)`, is evaluated and retained
  3. The sequence of 2-tuples are assembled into a list

NESTED LIST COMPREHENSIONS

- A list comprehension can have any number of *for-clauses* and *if-clauses*, freely-intermixed
- A *for-clause* must be first
- The clauses are evaluated from left to right
- ⇒ The Python Language Reference

```
# given a matrix 3x4 (see code)
[[row[i] for row in matrix] for i in range(4)]

transposed = []
for i in range(4):
    transposed.append([row[i] for row in matrix])
```

⇒ https://github.com/fpro-admin/lectures/blob/master/19/transpose.py

MAP(), FILTER()

- The `map()` and `filter()` built-in functions are handy functions for processing sequences without writing lengthy for-loops
- The idea of each is **to take a small function you write and apply it to all the elements of a sequence**, saving you from writing an explicit loop
- The implicit loop within each of these functions may be faster than an explicit **for** loop
- Additionally, each of these is a *pure function*, returning a result value
- This allows the results of the functions to be combined into complex expressions relatively easily

PROCESSING PIPELINE

- It is very, very common to apply a single function to every value of a list
- In some cases, we may apply multiple simple functions in a kind of "processing pipeline"

```
# NBA's players heights in (feet,inch)
heights = [(5,8), (5,9), (6,2), (6,1), (6,7)]

# convert (feet, inch) to inches
def ftin_2_in(ftin):
    feet, inches = ftin
    return 12.0*feet + inches

map(ftin_2_in, heights)
...

# now convert inches to meters
map(in_2_m, map(ftin_2_in, heights))
```

⇒ https://github.com/fpro-admin/lectures/blob/master/19/metric_sizes.py

MAP

- Create a new `iterator` from the results of applying the given *function* to the items of the the given *sequence*

```
map(function, sequence, [...])
```

- This function behaves as if it had the following definition:

```
def map(a_function, a_sequence):
    return [a_function(v) for v in a_sequence]
```

```
>>> list(map(int, ["10", "12", "14", 3.1415926, 5]))
[10, 12, 14, 3, 5]
```

- If more than one sequence is given, the corresponding items from each sequence are provided as arguments to the function (`None` used for missing values, as in `zip()`)

FILTER

- Return a `iterator` containing those items of sequence for which the given function is `True`
- If the function is `None`, return a list of items that are equivalent to `True`

```
filter(function, sequence)
```

- This function behaves as if it had the following definition:

```
def filter(a_function, a_sequence):
    return [v for v in a_sequence if a_function(v)]
```

```
>>> def over_2(m):
...     return m > 2.0
>>> list(filter(over_2, map(in_2_m, map(ftin_2_in, heights))))
[2.01]
```

⇒ https://github.com/fpro-admin/lectures/blob/master/19/filter.py

FILTER

- Return a `iterator` containing those items of sequence for which the given function is `True`
- If the function is `None`, return a list of items that are equivalent to `True`

```
filter(function, sequence)
```

- This function behaves as if it had the following definition:

```
def filter(a_function, a_sequence):
    return [v for v in a_sequence if a_function(v)]
```

```
>>> def over_2(m):
...     return m > 2.0
>>> list(filter(over_2, map(in_2_m, map(ftin_2_in, heights))))
[2.01]
```

⇒ https://github.com/fpro-admin/lectures/blob/master/19/filter.py

REDUCE

- **Removed in Python3!**
- Use `functools.reduce()` if you really need it
- However, 99 percent of the time an explicit `for` loop is more readable
- The idea is to apply the given function to an internal accumulator and each item of a sequence, from left to right, so as to reduce the sequence to a single value

```
def reduce(a_function, a_sequence, init= 0):
    r = init
    for s in a_sequence:
        r = a_function(r, s)
    return r
```

- built-in `sum()`, `any()` and `all()` are kinds of reduce functions

⇒ https://github.com/fpro-admin/lectures/blob/master/19/reduce.py

ZIP

- The `zip()` function interleaves values from two or more sequences to create a new sequence
- The new sequence is a sequence of tuples
- Each item of a tuple is the corresponding values from from each sequence
- If any sequence is too long, truncate it

```
zip(sequence, [sequence...])
```

- Here's an example:

```
list(zip( range(5), range(1,12,2) ))
[(0, 1), (1, 3), (2, 5), (3, 7), (4, 9)]
```

List Sorting

- Consider a list of tuples (that came from a spreadsheet `csv` file)

```
job_data = [
    ('121','Wyoming','NY',8722),
    ('123','Yates','NY',5094)
    ...
    ('001,'Albany','NY',162692),
    ('003','Allegany','NY',11986),
    ]
```

- Sorting this list can be done trivially with the `list.sort()` method

```
job_data.sort()
```

- This kind of sort will simply compare the tuple items in the order presented in the tuple
- In this case, the country number is first

SORTING WITH KEY EXTRACTION

- What if we want to sort by some other column, like state name or jobs?
- The `sort()` method of a list can accept a keyword parameter, `key`, that provides a key extraction function
- This function returns a value which can be used for comparison purposes
- To sort our `job_data` by the third field, we can use a function like the following:

```
def by_state(a):
    return a[1]

job_data.sort(key=by_state)
```

⇒ https://github.com/fpro-admin/lectures/blob/master/19/sort.py

THE LAMBDA

- The functions `map()`, `filter()` and the `list.sort()` method all use small functions to control their operations
- Instead of defining a function, Python allows us to provide a *lambda form*
- This is a kind of anonymous, one-use-only function body in places where we only need a very, very simple function
- A *lambda form* is like a defined function: it has parameters and computes a value
- The body of a *lambda*, however, can only be a single expression, limiting it to relatively simple operations

```
lambda a: a[0]*2+a[1]   # define a lambda

(lambda n: n*n)(5)      # define a lambda and call it
```

⇒ https://github.com/fpro-admin/lectures/blob/master/19/lambda.py

PARAMETERIZING A LAMBDA

- Sometimes we want to have a *lambda* with an argument defined by the “context” or “scope” in which the *lambda* is being used

```
>>> def timesX(x):
        return lambda a: x*a

>>> t2 = timesX(2)    # a function that multiplies the arg. by 2
>>> t2(5)
10

>>> t3 = timesX(3)    # a function that multiplies the arg. by 3
>>> t3(5)
15
```

⇒ https://github.com/fpro-admin/lectures/blob/master/19/lambda.py

EXERCISES

- Moodle activity at: LE18: FP with Collections