Programming Fundamentals

Effect-free Programming Style

João Correia Lopes

INESC TEC, FEUP

6 december 2018

GOALS

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

- Describe the *Effect-free programming styl*e: function calls have no *side effects* and variables are *immutable*
- Enumerate the Python language features that enables the programmer to adopt an Effect-free programming style

BIBLIOGRAPHY

- David Mertz, *Functional Programming in Python*, O'Reilly Media, 2015 [PDF]
- Andrew Kuchling, *Functional Programming HOWTO*, Python 3.6.7 documentation, Release 3.6.7, November 20, 2018 [HTML]

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

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/`**

Contents

Effect-free programming style

- Function calls have **no side effects** and variables are **immutable**
  - Do not use `global` and `nonlocal` statements
  - Take care about data types that are mutable
  - Do not use Input/Output

PYTHON & FUNCTIONAL PROGRAMMING

Python is most definitely not a "pure functional programming language"; side effects are widespread in most Python programs. That is, variables are frequently rebound, mutable data collections often change contents, and I/O is freely interleaved with computation.

It is also not even a "functional programming language" more generally.

However, Python is a multiparadigm language that makes functional programming easy to do when desired, and easy to mix with other programming styles.[1]

[1] David Mertz, Functional Programming in Python, O'Reilly Media, 2015

FUNCTIONAL PROGRAMMING

- In a functional program, input flows through a set of functions
- Each function operates on its input and produces some output
- Functional style discourages functions with *side effects* that *modify internal state* or make other changes that aren't visible in the function's return value
- Functions that have no side effects at all are called **purely functional**
- Avoiding side effects means not using data structures that get updated as a program runs; every function's output must only depend on its input

ADVANTAGES OF THE FUNCTIONAL STYLE

Why should you avoid objects (OOP) and side effects?

- There are theoretical (T) and practical (P) advantages to the functional style:
  - Formal provability (T)
  - Modularity (P)
  - Ease of debugging and testing (P)
  - Composability (P)

EFFECT-FREE CODE

The advantage of a pure function and side-effect free code is that it is generally easier to debug and test.

ADVANTAGES OF THE FUNCTIONAL STYLE

Why should you avoid objects (OOP) and side effects?

- There are theoretical (T) and practical (P) advantages to the functional style:
  - Formal provability (T)
  - Modularity (P)
  - Ease of debugging and testing (P)
  - Composability (P)

EFFECT-FREE CODE

The advantage of a pure function and side-effect free code is that it is generally easier to debug and test.

MODIFIERS VS PURE FUNCTIONS (RECAP)

- Functions which take lists as arguments and change them during execution are called **modifiers** and the changes they make are called **side effects**
- A **pure function** does not produce side effects
  - It communicates with the calling program only through parameters, which it does not modify, and a return value
- Is `double_stuff()` pure?

```
def double_stuff(values):
    """ Double the elements of values """
    for index, value in enumerate(values):
        values[index] = 2 * value
```

Iterators (recap)

- Iterators are an important foundation for writing functional-style programs
- An iterator is an object representing a stream of data and returns the data one element at a time
- Several of Python's built-in data types support iteration, the most common being lists and dictionaries
- An object is called **iterable** if you can **get an iterator for it**
- Python expects iterable objects in several different contexts, the most important being the `for` statement
- Iterators can be **materialised** as lists or tuples by using the `list()` or `tuple()` constructor functions
- Built-in functions such as `max()` and `min()` can take a single iterator argument
- The `in` and `not in` operators also support iterators
- Note that you can **only go forward in an iterator**; there's no way to get the previous element, reset the iterator, or make a copy of it

IMPERATIVE PYTHON PROGRAMS

- "In typical imperative Python programs[2] a block of code generally consists of some outside loops (`for` or `while`), assignment of state variables within those loops, modification of data structures like `dicts`, `lists`, and `sets` (or various other structures, either from the standard library or from third-party packages), and some branch statements (`if/elif/else` or `try/except/finally`)."
- The imperative flow control described in the last paragraph is much more about the "how" than the "what" and **we can often shift the question**

[2] Including those that make use of classes and methods to hold their imperative code.

Comprehensions

- Using comprehensions is often a way both to make code more compact and to *shift our focus from the "how" to the "what"*
- A comprehension is an expression that uses the same keywords as loop and conditional blocks, but inverts their order to **focus on the data rather than on the procedure**
- Simply changing the form of expression can often make a surprisingly large difference in how we *reason about code* and how easy it is to understand it
- Python includes: *List comprehensions*, *Generator comprehensions*, *Set comprehensions*, and *Dictionary comprehensions*

MENTAL SHIFT

- The *ternary operator* also performs a similar restructuring of our focus, using the same keywords in a different order
- For example, if our original code was:

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

- Somewhat more compactly we could write this as:

```
collection = [d if condition(d) else modify(d)
              for d in data_set]
```

Generators

- *Generator comprehensions* have the same syntax as list comprehensions — other than that there are no square brackets around them (but parentheses are needed syntactically in some contexts, in place of brackets)
- They are also *lazy*
- That is to say that they are merely a description of "how to get the data" that is not realised until one explicitly asks for it, either by calling `.next()` on the object, or by looping over it

```
    log_lines = (line for line in read_line(huge_log_file)
                      if complex_condition(line))
```

⇒ https://github.com/fpro-admin/lectures/blob/master/20/generators.py

RECURSION (RECAP)

- Functional programmers often put weight in **expressing flow control** through recursion rather than through loops
- Done this way, we can **avoid altering the state** of any variables or data structures within an algorithm, and more importantly get more at the "what" than the "how" of a computation
- In the cases where recursion is just "iteration by another name", iteration is more "Pythonic"
- Where recursion is compelling, and sometimes even the only really obvious way to express a solution, is when a problem offers itself to a "divide and conquer" approach (i.e., a problem can readily be partitioned into smaller problems)

⇒ `https://github.com/fpro-admin/lectures/blob/master/20/factorialR.py`

QUICKSORT

- For example, the *quicksort algorithm* is very elegantly expressed without any state variables or loops, but wholly through recursion

```python
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
```

⇒ https://github.com/fpro-admin/lectures/blob/master/20/quicksort.py

CALLABLES

- The emphasis in functional programming is on calling functions
- Python actually gives us several different ways to create functions, or at least something very *function-like* (i.e., that can be called):
  1. Regular functions created with `def` and given a name at definition time
  2. Anonymous functions created with *lambda*
  3. *Generator* functions
  4. *Closures* returned by function factories
  5. Instances of classes that define a `__call()__` method
  6. Static methods of instances, either via the `@staticmethod` decorator or via the class `__dict__`

CALLABLES

- The emphasis in functional programming is on calling functions
- Python actually gives us several different ways to create functions, or at least something very *function-like* (i.e., that can be called):
  1. Regular functions created with `def` and given a name at definition time
  2. Anonymous functions created with *lambda*
  3. *Generator* functions
  4. *Closures* returned by function factories
  5. Instances of classes that define a `__call()__` method
  6. Static methods of instances, either via the `@staticmethod` decorator or via the class `__dict__`

Named Functions and Lambdas (recap)

- The most obvious ways to create callables in Python are named functions and lambdas
- In most cases, lambda expressions are used within Python only for callbacks and other uses where a simple action is inlined into a function call

```
>>> def hello1(name):
.....     print("Hello", name)
.....
>>> hello2 = lambda name: print("Hello", name)
>>> hello1('John')
Hello John
>>> hello2('John')
Hello John

>>> hello3 = hello2   # can bind func to other names
>>> hello3.__qualname__
'<lambda>'
```

⇒ https://github.com/fpro-admin/lectures/blob/master/20/callable.py

CAVEATS, LIMITS, AND DISCIPLINE

FUNCTIONAL PROGRAMMING STYLE

In most cases, one only leaks state intentionally, and creating a certain subset of all your functionality as pure functions allows for cleaner code

- One of the reasons that functions are useful is that they **isolate state lexically**
- This is a limited form of nonmutability in that (by default) nothing you do within a function will bind state variables outside the function
- This guarantee is very limited in that both the `global` and `nonlocal` statements explicitly allow state to “leak out” of a function
- Moreover, many **data types are themselves mutable**, so if they are passed into a function that function might change their contents
- Furthermore, doing **I/O** can also change the “state of the world” and hence alter results of functions[3]

[3] E.g., by changing the contents of a file or a database that is itself read elsewhere.

CLOSURES AND CALLABLE INSTANCES

- A **closure** is "operations with data attached" (putting operations and data in the same object)
- Closures emphasise immutability and pure functions
- A Closure is a function object that remembers values in enclosing scopes even if they are not present in memory

```
def make_adder(n):
    def adder(m):
        return m + n
    return adder

add5_f = make_adder(5)   # "functional"

>>> add5_f(10)
15
```

⇒ https://github.com/fpro-admin/lectures/blob/master/20/closure.py

Generator Functions

- A special sort of function in Python is one that contains a `yield` statement, which turns it into a generator
- What is returned from calling such a function is not a regular value, but rather an **iterator** that produces a sequence of values as you call the `next()` function on it or loop over it
- For example, see the code for “Simple lazy Sieve of Eratosthenes” in `get_primes()`
- Every time you create a new object with `get_primes()` the iterator is the same infinite lazy sequence

⇒ https://github.com/fpro-admin/lectures/blob/master/20/get_primes.py

LAZY EVALUATION

ITERATORS

Iterators are lazy sequences rather than realised collections

- Python does not quite offer *lazy data structures* in the sense of a language like Haskell
- However, use of the iterator protocol and Python's many built-in or standard library iteratables, accomplish much the same effect as an actual lazy data structure
- The easiest way to create an iterator — that is to say, a lazy sequence — in Python is to define a **generator function**
- Well, technically, the easiest way is to use one of the many *iterable objects* already produced by built-ins or the standard library rather than programming a custom one at all
- The module `itertools` is a collection of very powerful, and carefully designed, functions for performing *iterator algebra*

HIGHER-ORDER FUNCTIONS

- Higher-order functions (often abbreviated as "HOF") provide building blocks to express complex concepts by combining simpler functions into new functions
- In general, a higher-order function is simply a function that takes one or more functions as arguments and/or produces a function as a result
- It is common the think of `map()`, `filter()`, and `functools.reduce()` as the most basic building blocks of higher-order functions
- Almost as basic as map/filter/reduce as a building block is currying
  - In Python, currying is spelled as `partial()`, and is contained in the `functools` module
  - This is a function that will take another function, along with zero or more arguments to pre-fill, and return a function of fewer arguments that operates as the input function would when those arguments are passed to it

EQUIVALENCIES

- The built-in functions `map()` and `filter()` are equivalent to comprehensions (especially now that generator comprehensions are available)

```python
# Classic "FP-style"
transformed = map(tranformation, iterator)

# Comprehension
transformed = (transformation(x) for x in iterator)

# Classic "FP-style"
filtered = filter(predicate, iterator)

# Comprehension
filtered = (x for x in iterator if predicate(x))

from functools import reduce
total = reduce(operator.add, it, 0)

# total = sum(it)
```

EXERCISES

- Moodle activity at: LE20: Effect-free programming style