F21SC Industrial Programming: Functional Programming in Python

Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh

Semester 1 2017/18

[0]No proprietary software has been used in producing these slides

What is a functional language?

The distinctive feature of **pure** functional languages is their **referential transparency**.

Definition (Stoy, 1977)

The only thing that matters about an expression is its value, and **any subexpression can be replaced by any other equal in value**. Moreover, the value of an expression is, within certain limits, the same wherever it occurs.

Implications:

- Two expressions are *equal* if they have the same value, e.g. sin(6) = sin(1+5).
- Value-based equality enables *equational reasoning*, where one expression is substituted by another of equal value, e.g. f(x) + f(x) = 2*f(x)
- Scope matters: if x = 6, then sin(x) = sin(6) = sin(1+5)

Properties of functional languages

- **Computations in functional languages are free of side effects.** See above: these languages are **referentially transparent**.
- **Computations in functional languages are state-less.** This excludes the almost ubiquitous pattern in imperative languages of assigning first one, then another value to the same variable to track the program state.
- **Functions are first class (objects).** Everything you can do with "data" can be done with functions themselves (such as passing a function to another function).
- **Recursion is used as a primary control structure.** In some languages, no other "loop" construct exists.

Properties of functional languages (cont'd)

- There is a focus on **list processing**. Lists are often used with recursion on sublists as a substitute for loops.
- Functional programming either discourages or outright disallows statements, and instead works with the evaluation of expressions (in other words, functions plus arguments). In the pure case, one program is one expression (plus supporting definitions).
- Functional programming focusses on **what** is to be computed **rather than how** it is to be computed.
- Much functional programming utilises "higher order" functions (in other words, functions that operate on functions that operate on functions).

Functional languages are free of state

Goal: Create a list of all square values of some integer numbers.
Imperative solution[1]:

Example

```
def square(x):
    return x*x

input = [1, 2, 3, 4]
output = []
for v in input:
    output.append(square(v))
```

NB: the contents of the list `output` changes as you iterate over `input`; to understand the program, you have to follow the control-flow

[1] From https://marcobonzanini.com/2015/06/08/functional-programming-in-python/

Functional languages are free of state (cont'd)

Functional solution[2]:

Example

```
def square(x):
    return x*x

input = [1, 2, 3, 4]
output = map(square, input)
```

NB: in this version, there is **no internal state**; the result list is defined in one go (bulk operation); you only need to understand the operation on each element.

[2] From https://marcobonzanini.com/2015/06/08/functional-programming-in-python/

Functionally inspired constructs in Python

1. List comprehensions
2. Set and dictionary comprehensions
3. Libraries of higher-order functions (`map`, `reduce`, `zip`)
4. Iterators (also called **"lazy data structures"**)

Material from **Functional Programming in Python**, by David Mertz, O'Reilly Media, 2015.

List comprehensions

These change the way how you **think** about the data-structure: focus on **what** the collection is, rather than **how** it is constructed.

Example

combs = [(x, y) for x in [1,2,3] for y in [1,2,3] if x != y]

This is equivalent to this more verbose code

Example

```
combs = []
for x in [1,2,3]:
    for y in [1,2,3]:
        if x != y:
            combs.append((x, y))
```

Generator comprehensions

These describe "how to get the data", that is **not realised** until one explicitly asks for it. They implement **lazy data structures**.

Example

```
with open(file,"r") as f:
   xs = (line for line in f
              if  re.search('"event_type":"read"', line))
```

NB: in this example, f is a generator, iterating over the file; you can't directly ask for its length

[2] See this sample source code

Generator comprehensions (cont'd)

A traditional, more verbose, version:

Example

```
def proc_file(file):
   """Find read events in an issuu log file."""
   f = open(file,"r")
   for line in f:
       if re.search('"event_type":"read"', line):
           yield line
   f.close()
```


Set and dictionary comprehensions

In the same way that list comprehensions are used to define a list in a bulk operation, set or dictionary comprehensions can be used to define sets or dictionaries in one construct.

Example

```
{ i:chr(48+i) for i in range(10) }
```

This defines a dictionary, mapping digits (0–9) to their character codes.

Libraries of higher-order functions

The most common higher-order functions are[3]:

- `map`: perform the same operation over every element of a list
- `filter`: selects elements from a list, to form a new list
- `reduce` (in module `functools`): do a pair-wise combination over all elements of a list
- `zip`: takes one element from each iterable and returns them in a tuple
- `any`: checks whether **any** element of a list fulfills a given predicate
- `all`: checks whether **all** elements of a list fulfills a given predicate
- `takewhile`: returns elements for as long as the predicate is true
- `dropwhile`: discards elements while the predicate is true
- `groupby`: collects all the consecutive elements from the underlying iterable that have the same key value

NB: in Python 3.x, all these functions are **iterators**; therefore, usage is different from Python 2.x (see the examples on the next slides)

[3] See https://docs.python.org/3.4/howto/functional.html

Libraries of higher-order functions

- `filter(test, sequence)` returns a sequence, whose elements are those of `sequence` that fulfill the predicate `test`. E.g.

  ```
  filter(lambda x: x % 2 == 0, range(10))
  ```

- `map(f, sequence)` applies the function `f` to every element of `sequence` and returns it as a new sequence.

  ```
  map(lambda x: x*x*x, range(10))
  map(lambda x,y: x+y, range(1,51), range(100,50,-1))
  ```

- `reduce(f, [a1,a2,a3,...,an])` computes `f(...f(f(a1,a2),a3),...,an)`

  ```
  reduce(lambda x,y:x*y, range(1,11))
  ```

- `reduce(f, [a1,a2,...,an], e)` computes `f(...f(f(e,a1),a2),...,an)`

More higher-order examples

Demo

[3] See this sample source code

Recursion vs. Iteration

The following two versions of factorial are equivalent:

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
```


Recursion vs. Iteration

As a footnote, the fastest version I know of for factorial() in Python is in a functional programming style, and also expresses the "what" of the algorithm well once some higher-order functions are familiar:

Example

```
from functools import reduce
from operator import mul

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

Recursion vs. Iteration

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

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
```

Some names are used in the function body to hold convenient values, but they are never mutated

The concept of "**lazy data structures**"

- We typically think of a data structure as **fully expanded data**
- In most cases, this matches the representation of data in memory
- However, sometimes we **do not want fully expanded data**, but still use it as a normal data structure
- E.g. when working with large data sets, we just want to iterate over the data
- A data structure that is only expanded if and when it is used, is a **lazy data structure**
- Python's **generators** (and to some degree **iterators**) allow to use lazy data structures
- In Python 3, iterators are the preferred mechanism for constructing data structures

Simple lazy Sieve of Eratosthenes

Example

```
def get_primes():
    "Simple lazy Sieve of Eratosthenes"
    candidate = 2
    found = []
    while True:
        if all(candidate % prime != 0 for prime in found):
            yield candidate
            found.append(candidate)
        candidate += 1

primes = get_primes()
#
print(next(primes), next(primes), next(primes))
# (2, 3, 5)
for _, prime in zip(range(10), primes):
    print(prime, end=" ")
```


Summary

- Python borrows many of its advanced language features from functional languages
- List and generator **comprehensions**, for concise, bulk operations on data structures
- **Higher-order functions** to encode commonly occuring compute structures
- **Generators** (and iterators) to get the benefits of "lazy data structures"