Functional Programming

Christopher R. Genovese

Department of Statistics & Data Science

Tue 06 Oct 2025
Session #13

Plan

First-Class Entitites

Plan

First-Class Entitites

Core Concepts of Functional Programming

Plan

First-Class Entitites

Core Concepts of Functional Programming

Higher-Order Functions

Plan

First-Class Entitites

Core Concepts of Functional Programming

Higher-Order Functions

Activities

Announcements

- Questions?
- **Homework**:
  - **classification-tree-basic** assignment due Thu 23 Oct.
  - Push outstanding mini-assignments when complete

Programming with *Verbs*

As discussed earlier, we can think of OOP as *programming with nouns*, with objects (the nouns) as the central abstraction.

Today, we look at a different paradigm: *programming with verbs*, where *functions* are the central abstraction. This is **functional programming**.

Programming with *Verbs*

As discussed earlier, we can think of OOP as *programming with nouns*, with objects (the nouns) as the central abstraction.

Today, we look at a different paradigm: *programming with verbs*, where *functions* are the central abstraction. This is **functional programming**.

Another useful contrast is with *imperative programming*, where we tell the computer how to do each step of our computation:

```
x = 2
db = setup_database(x)
update(db)
output_results(db)
```

In contrast, with functional programming, we view the computation as the evaluation of a function – itself composed of other functions through which we thread data. This leads to a more *declarative programming*, where we tell the computer what to do rather than explicitly how to do it.

A Challenge Preview

Suppose that cows of the same breed get into an argument with each other if they are standing too close together. Two cows of the same breed are "crowded" if their positions within a line of n cows differ by no more than k, where $1 \leq k < n$.

Given a value of k and a sequence representing the breed IDs of a line of cows, compute the maximum breed ID of a pair of crowded cows.

If there are no crowded cows, return -1.

Challenge: Do not use loops or any imperative constructs.

Plan

First-Class Entitites

Core Concepts of Functional Programming

Higher-Order Functions

Activities

Paradigm Shift

A **programming paradigm** is a framework of organizing concepts and abstractions that guide how we design and build software and how we think about solving problems.

Two familiar and commonly used paradigms are:

- *object-oriented programming* :: the *class* is the fundamental unit of abstraction - data and code are wrapped together, encapsulated, on objects that represent the *nouns* of our system.
- *imperative programming* :: a program is a sequence of *commands* and *actions* for the computer to perform that successively update program state. An imperative program describes how to perform a task.
- *data-oriented programming*: a description of *data* is the fundamental unit of abstraction, with code separated from data and data represented with immutable, generic data structures.
- *functional programming*: *functions* are the fundamental unit of abstraction, and a program is the composition of functions.

Today, we start a deep dive into **functional programming**.

First-Class Entities

An entity is **first class** when it can be:

- created at run-time,
- passed as a parameter to a function,
- returned from a function, or
- assigned into a variable.

A first class entity typically has built-in support within a language and runtime environment.

Example: Fixed-width integers are first-class entities in all common languages.

- Created: `101`
- Passed: `max(11, -4)`
- Returned: `return 7`
- Assigned: `i = 17`

Similarly with floating-point numbers, booleans, strings, and arrays. More specialized examples:

- Arbitrary precision numbers (Clojure, Mathematica, Java)
- Hash Tables/Associative Arrays/Tables (Python, Clojure, Perl, Ruby, Lua)
- Raw Memory Locations (C, C++)
- Classes/Objects (Python, Ruby, C++, Java, ...)
- Types (Idris, Agda, TL1)
- Unevaluated Code (Common Lisp, Racket, Scheme, Clojure, Rust, R, Haskell)
- **Functions** (Python, R, Haskell, Unison, Clojure, OCaml, Rust, JavaScript, ...)

The first-class entities in a language shape a languages central idioms and abstractions.

Plan

First-Class Entitites

Core Concepts of Functional Programming

Higher-Order Functions

Activities

Composability

Composability is a fundamental aspiration in software design. We want the parts of our system to interoperate and combine as freely as possible. This offers power, expressiveness, reusability, and many other benefits.

Examples of failed composability:

```
query = 'select x, y, z from foo where x = ?'
cursor = sql.execute(query, 10)
```

If we want to modify the query, say adding a column or a constraint, how do we do that.

Printer control (Runar Bjarnason)

Generalization: Algebraic abstraction

Why FP?

- Functions are simple, easily composed abstractions
- Easy to express many ideas with very little code
- Easier to reason about – and test – a functional program
- Avoids the significant complexity of mutating state
- Effective Concurrency/Parallelism by avoiding mutable state
- Focuses on data flow
- Efficiently exploits recursive thinking
- Fast development cycle
- Encourages code reuse
- Isolate **side effects**

Languages designed for FP: Clojure, Haskell, OCaml, Scala, Idris, Lean

Languages with great FP support: Rust, R, Common Lisp, Julia, Ruby, JavaScript

Decent FP Support: Python

Languages adding many FP features: C++, Java

Core Concepts of FP

- Functions are **first-class** entities
- Functions are **pure** (whenever possible)
- Data is **immutable** (whenever possible or non-locally)
- Programs have **declarative** structure (as much as possible)
- Exploit **laziness** when appropriate
- **Referential transparency**

First-Class Functions

Functions are data too! Use cases:

1. Parameterized strategies
2. Generic operations - abstracting operations across a range of types
3. Dynamically-defined operations based on parameters and data
4. Combine multiple pieces into useful aggregate functions

First-Class Functions (cont'd)

```
apply(M, ACROSS_COLS, function(x) { max(x[!is.na(x) && x != 999]) })

pairwise.distances(X, metric = function(x, y) { max(abs(x - y)) })

integrate(lambda x: x*x, 0, 1)

// Abstracting Array Iteration
function forEach(array, itemAction) {
    for ( var i = 0; i < array.length; i++ ) {
        itemAction(array[i])
    }
}
forEach(["R", "SAS", "SPSS"], console.log);
forEach(["R", "SAS", "SPSS"], store);
forEach(["R", "SAS", "SPSS"], function(x) {myObject.add(x)});
```

First-Class Functions

```python
def fwd_solver(f, z_init):
  "Fixed-point solver by forward iteration"
  z_prev, z = z_init, f(z_init)
  while np.linalg.norm(z_prev - z) > 1e-5:
    z_prev, z = z, f(z)
  return z

def newton_solver(f, z_init):
  "Newton iteration fixed-point solver"
  f_root = lambda z: f(z) - z
  g = lambda z: z - np.linalg.solve(jax.jacobian(f_root)(z), f_root(z))
  return fwd_solver(g, z_init)
```

First-Class Functions (cont'd)

```clojure
;; *Markov Chain Monte Carlo (MCMC)* is a simulation method
;; where we create a Markov chain whose *limiting distribution*
;; is a distribution from which we want to sample.

(defn mcmc-step
  "Make one step in MH chain, choosing random move."
  [state moves]
  (let [move (random-choice moves)]
    (metropolis-hastings-step (move state))))

(def chain
  (mcmc-sample initial-state
               :select (mcmc-select :burn-in 1000 :skip 5)
               :extract :theta1
               :moves [(univariate-move :theta1 random-walk 0.1)
                       (univariate-move :theta2 random-walk 0.4)
                       (univariate-move :theta3 random-walk 0.2)]))
(take 100 chain)
```

Pure Functions

What does this code do?

```
x = [5]
process( x )
x[0] = x[0] + 1
```

A function is **pure** if it:

- always returns the same value when you pass it the same arguments
- has no /observable/ side effects

What are "side effects"?

- Changing global variables
- Printing out results
- Modifying a database
- Editing a file
- Generating a random number
- ... anything else that changes the rest of the world outside the function.

Pure Functions (cont'd)

```
pure <- function(x) {
    return( sin(x) )
}
global.state <- 10

not.pure <- function(x) {
    return( x + global.state )
}
also.not.pure <- function(x) {
    return( x + rnorm(1) )
}
another.not.pure <- function(x) {
    save_to_file(x, "storex.txt")
    return( x + rnorm(1) )
}
u <- 10
and.again <- function(x) {
    ...
    print(...)              # Input/Output
    z <- rnorm(n)           # Changing internal state
    my.list$foo <- mean(x)  # Mutating objects' state
    u <<- u + 1             # Changing out-of-scope values
    ...
}
```

Pure Functions (cont'd)

```
def foo(a):
    a[0] = -1

    return sum(a)

x = [1, 2, 3]

sum_of_x = foo(x)

x  #=> [-1, 2, 3]
```

Pure Functions (cont'd)

Benefits of pure functions

1. deterministic and mathematically well-defined
2. easy to reason about
3. easy to test
4. easy to change
5. can be composed and reused
6. can be run simultaneously in concurrent processes
7. can be evaluated lazily

Aside: What is Pure anyway?

An expression `e` is **referentially transparent** if for all programs `p`, every occurrence of `e` in `p` can be replaced with the result of *evaluating* `e` without changing the result of evaluating `p`.

A function `f` is **pure** if the expression `f(x)` is referentially transparent for all referentially transparent `x`.

Calculations, Actions, and Data

- **Action** :: results depend on when they are called, how many times they are called, or context in which they are called (aka **Effects**)
- **Calculations** :: operations results that do not affect the world when they run, processing inputs to output directly, independent of context or timing
- **Data** :: Facts about events, a record of something that happened, encodes meaning in structure

It is useful to practice distinguishing among these three categories with your code.

We prefer calculations (pure functions) where possible and keep a clear delineation.

Immutable Data

This is a common pattern in imperative programming:

```
a = initial value
for index in IndexSet:
    a[index] = update based on index, a[index], ....
return a
```

Each step in the loop updates – or *mutates* – the state of the object.

The familiarity of this operation obscures the **tremendous increase** in complexity it produces. Mutability couples different parts of your code and across time. Greatly complicates parallism and concurrency.

Immutable Data (cont'd)

Immutable (persistent) data structures do not change once created.

- We can pass the data to anywhere (even simultaneously), knowing that it will maintain its meaning.
- We can maintain the history of objects as they transform.
- With pure functions and immutable data, many calculations can be left to when they are needed (laziness).

These data structures achieve this with good performance by clever **structure sharing**. (Eg: `pyrsistent` library in Python, `immer` and `immutable.js` in JavaScript, built-in in Clojure, `rstackdeque` in R, ...) (cf. Rust)

Immutable Data (cont'd)

Favoring immutable data and pure functions makes values and transformations, rather than actions, the key ingredient of programs. So FP prefers expressions over statements

```
(foo (if (pred x) x (transform x)) y)

if pred(x):
    y = x
else:
    y = transform(x)
foo(y)
```

Declarative Structure

A **declarative** program tells us what the program produces *not* how to produce it.

```
qsort []    = []
qsort l@(x:xs) = qsort small ++ mid ++ qsort large
  where
    small = [y | y <- xs, y < x]
    mid      = [y | y <- l, y == x]
    large = [y | y <- xs, y > x]
```

Programming languages or packages often provide ways to thread data through multiple functions to clearly express the flow of operations:

```
tokenize <- function(line) {
    line %>%
        str_extract_all("([A-Za-z][-A-Za-z]+)") %>%
        unlist %>%
        sapply(tolower) %>%
        as.character
}
## much clearer than:
## as.character(sapply(tolower, unlist(str_extract_all("([A-Z]...)", line))))
```

```
(defn tokenize [line]
  (->> line
       (re-seq "([A-Za-z][-A-Za-z]+)")
       (map lower-case)))
```

Closures

Closures are functions with an environment – variables and data – attached. The environment is persistent, private, and hidden. This is a powerful approach for associating state with functions that you pass into other functions. (In fact, an entire OOP system could be built from closures.)

```
counter <- function(start=0, inc=1) {
    value <- start

    return(function() {
        current <- value
        value <<- value + inc
        return(current)
    })
}
```

```
cc <- counter(10, 2)
dd <- counter(10, 2)
cc() # 10
cc() # 12
cc() # 14...
value # Error: object 'value' not found
dd() # 10
```

```
kernel_density <- function(data, bandwidth) {
    return(function(x) {
        ## calculate the density here
        for (row in 1:nrow(data)) {
             ## calculate some stuff
        }
        return(density)
    })
}
d <- kernel_density(data, 4)
d(3) #=> returns density at 3
```

Plan

First-Class Entitites

Core Concepts of Functional Programming

Higher-Order Functions

Activities

Higher-Order Functions

Higher-order functions are functions that take other functions as arguments and/or return a function as its result.

These are broadly useful. For instance, we can parameterize computational strategies in our methods, create functions to do specialized calculations, and represent, and represent a broad class of computations.

Higher-Order Functions

As examples, here are some commonly used higher-order functions. Below, we use `f : Type -> Type` to represent a container/collection of values of a specified type, e.g., `f = List`, `f = Vector n`, `f = Maybe`, or `f = RoseTree`.

- `map : (a -> b) -> f a -> f b`
  The `map` operation calls a function on every element of a collection and returns a collection of the same type and shape with the results.
- `filter : (a -> Bool) -> f a -> f a`
  The `filter` operation uses a predicate to select elements of a collection, keeping the elements for which the predicate is true.
- `mapMaybe : (a -> Maybe b) -> f a -> f b`
  The `mapMaybe` operation combines `map` and `filter`, keeping the transformed values `v` for which the given function returns `Some v`.
- `fold : (acc -> elt -> acc) -> acc -> f elt -> acc`
  The `fold` operation implements the fold pattern.

Map

The `map` operation takes a function and a collection and calls the function for each successive element of the collection, producing a new collection out of the results. For example, in R:

```r
square <- function(x) x^2
Map(square, 1:10)   #=> list(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
```

or in Python:

```python
list(map(lambda x: x*x, range(1, 11)))
  #=> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

using an *anonymous* function, declared with the `lambda` keyword, that has no name.

Map

map can usually do more than this. In most languages, it can take /multiple sequences/ and a function that takes multiple arguments. It then calls the function with successive elements from each sequence as its arguments, producing a sequence of the results. For example, in R

```
Map(`-`, 1:10, 10:1)   #=>  list(-9, -7, -5, -3, -1, 1, 3, 5, 7, 9)
```

and in Python

```
list(map(lambda x, y: x - y, range(1, 11), range(10, 0, -1)))
   #=> [-9, -7, -5, -3, -1, 1, 3, 5, 7, 9]
```

We can use map to replace a common pattern we see over and over in code processing data:

```
transformed_data <- c()

for (ii in 1:length(data)) {
    transformed_data <- c(transformed_data, do_stuff_to(data[ii]))
}

## becomes

transformed_data <- Map(do_stuff_to, data)
```

The map operation expresses our meaning more clearly and concisely, and is more efficient to boot (since it does not repeatedly copy the transformed_data).

Filter

The `filter` operation takes a predicate (a function returning a boolean) and a collection. It calls the predicate for each element, and returns a new collection containing only those elements for which the predicate returns true. For example, in R:

```
is_odd <- function(x) { (x %% 2 != 0) }

Filter(is_odd, 1:10) #=>  c(1, 3, 5, 7, 9)
Filter(is_odd, as.list(1:10)) #=>  list(1, 3, 5, 7, 9)
```

Notice that each result is the same type as the collection `Filter` was given.

In Python,

```
filter(lambda x: x % 2 != 0, range(1, 11)) #=> [1, 3, 5, 7, 9]
```

We can always combine a `map` with a `filter`, applying a function to only those elements matching a predicate. The composability of these operations is a major advantage, and much easier to deal with than building complicated loops over data manually.

Fold

The `fold` operation is a general way to process the elements of a sequence. We could use it to build `map`, `filter`, or many other interesting operations. (This is sometimes called `reduce`.)

`fold` takes a "folding" function, an *accumulator*, and a sequence. The folding function takes the accumulator and one sequence element, returning an updated accumulator.

For example, suppose we want to add up the numbers in a list, but keep separate sums of the odd numbers and the even numbers. In R:

```r
parity_sum <- function(accum, element) {
    if ( element %% 2 == 0 ) {
        list(accum[[1]] + element, accum[[2]])
    } else {
        list(accum[[1]], accum[[2]] + element)
    }
}
Reduce(parity_sum, 1:10, list(0,0))  #=> list(30, 25)
```

In Python:

```python
def parity_sum(acc, x):
  even, odd = acc
  if x % 2 == 0:
      return [even + x, odd]
  else:
      return [even, odd + x]
reduce(parity_sum, range(1,11), [0,0])  #=> [30, 25]
```

Plan

First-Class Entitites

Core Concepts of Functional Programming

Higher-Order Functions

Activities

FP Intro Activities

Choose one of the tasks below at an appropriate challenge level. **Do not use loops or any imperative constructs**.

1. Cow Proximity

   Suppose that cows of the same breed get into an argument with each other if they are standing too close together. Two cows of the same breed are "crowded" if their positions within a line of n cows differ by no more than k, where $1 \leq k < n$.

   Given a value of k and a sequence representing the breed IDs of a line of cows, compute the maximum breed ID of a pair of crowded cows.

   If there are no crowded cows, return -1.

2. Write a function that sums a list of numbers.

3. Write a function that takes a list/vector of integers and returns only those elements for which the *preceding* element is negative.

4. Write a function that takes in a string and returns true if all square `[ ]`, round `( )`, and curly `{ }` delimiters are properly paired and legally nested, or returns false otherwise. For example, `[(a)]{[b]}` is legally nested, but `{a}([b)]` is not.

   Hint: What data structure can you use to track the delimiters you've seen so far?

FP Intro Activities

Choose one of the tasks below at an appropriate challenge level. **Do not use loops or any imperative constructs**.

5. Write a function `roman` that parses a Roman-numeral string and returns the number it represents. You can assume that the input is well-formed, in upper case, and adheres to the "subtractive principle" (link). You need only handle positive integers up to MMMCMXCIX (3999), the largest number representable with ordinary letters. You can also define constant objects or expressions (e.g., a dictionary to map characters like X, I, V, etc. to their corresponding value).

6. Write a function `chain` that takes as its argument a list of functions, and returns a new function that applies each function in turn to its argument. (This requires a language with first-class functions, like R or Python.)

   For example,

   ```
   import math

   positive_root = chain([abs, math.sqrt])

   positive_root(-4)    #=> 2.0
   ```

FP Intro Activities

Choose one of the tasks below at an appropriate challenge level. **Do not use loops or any imperative constructs**.

7. Write a function `partial` that takes a function and several arguments and returns a function that takes additional arguments and calls the original function with all the arguments in order. For example,

```
foo <- function(x, y, z) { x + y + z }
bar <- partial(foo, 2)

bar(3, 4) #=> 2 + 3 + 4 = 9
```

Note: You don't actually need `map`, `reduce`, or `filter` for this, but it's still good to know how to do it.

THE END