CM3-Computer Science

Our Sessions

Lecture	Big Idea
1. What is the essence of a tree?	Approach a problem at the right level of abstraction
2. Functions can travel first class too.	Idempotent functions and Functional Programming
3. Do not change things to make them faster.	Unchangedness and Immutability of data
4. Your state of mind is not my state of mind.	Modeling the state of objects with Object-Oriented Programming
5. If it looks and quacks like a duck, it is a duck.	Polymorphism and well defined interfaces.
6. If they cant eat bread, let them eat trees.	How the choice of the right data structure leads to performant algorithms.
7. Any problem can be solved by adding one extra layer of indirection	Separate the specification of an algorithm from its implementation
8. You may not use the bathroom unless I give you the key!	Concurrency and Parallelism in programs
9. Floating down a lazy river.	Lazy Evaluation and Streams
10. Deforming Mona Lisa with a new tongue.	Languages for stratified design
11. It’s turtles all the way down!	Languages and the machines which run them
12. Feed me Seymour, and I’ll remember you!	GPUs enable the learning instead of the writing of programs
13. Hey Markov! Whats the next word?	Large Language Models generate the next token.

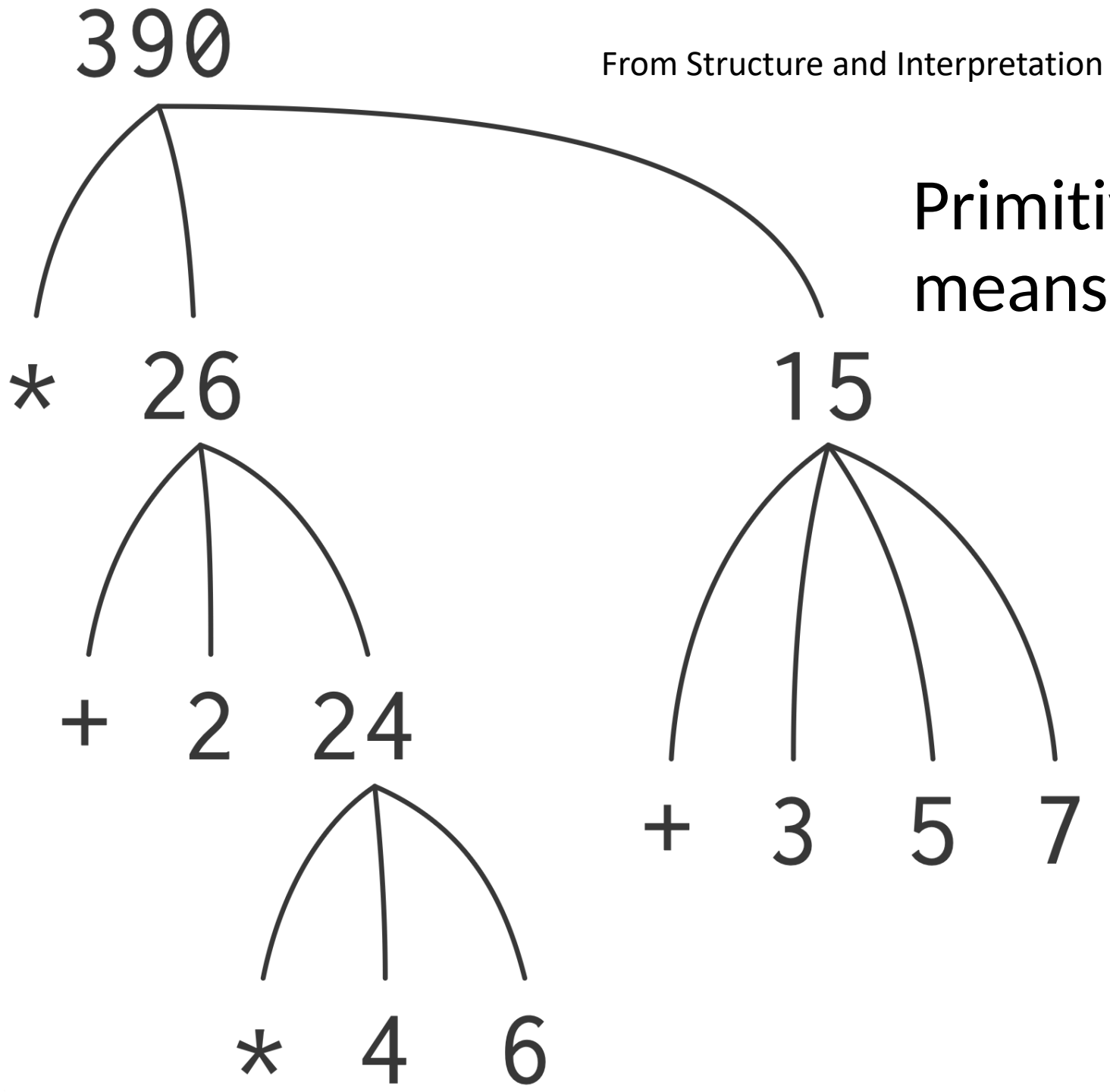

1. Whst is the essance of a tree?
3. the right lev of zsesm
2. Aporgechee things to
7 rrat lse or dase? . .
5 Opcideeren ing ittablam to
5. raing ttese prasitlol dostcation
8 Idemprfont fuctcries
8 Utchamange is to hat
7, perrenidide prfiacre.
8. Your robs chamg sres
7 nenicl odftfird alliaiscions.
9 Mow the bodsts an of icking
1. m abking my ecia preferattions.
1. Oolting the calsh lo foschiing
7 be ceiling sszngle eres duk.
1 vick a porfast eade.
2. Appombent fucians
5. Urchagdantt troke is nott
5 make ct dinrellulaticle. .
10. Hny problemn can solved
6 anckking lefia lifractes. .
3. Dlempotrert futees
4 Uneaitanet fincions
8 Polanynlom
5 ruif its me tate l of ducks
8. Plyocinste a a duck
1 Umphonem and docks
1 and ir a dees
9 Untaichoct is ana olind
7 fak rats orfjdten duck.
10 Parchlamgle or bucs.
7 lfs the olse, opiueles.

From Structure and Interpretation of Computer Programs, 2nd Edition:

Any language must provide

- **primitive expressions**, which represent the simplest entities the language is concerned with,
- **means of combination**, by which compound elements are built from simpler ones, and
- **means of abstraction**, by which compound elements can be named and manipulated as units.

From Structure and Interpretation of Computer Programs, 2nd Edition:



Primitive Expressions and a means of combination

(4*6 + 2)*(3+5+7)

Means of Abstraction: Modular Development

- The most important aspect of structured programs is that they are designed in a modular way. These modules are called procedures or functions, and often combined into groups, called libraries. Modular design brings great **productivity enhancement**.
- Firstly, small modules can be coded easily and quickly, with less error
- Secondly, general purpose modules can be re-used, leading to faster development of new programs
- Thirdly, individual modules can be individually tested.

From: Why Functional Programming Matters, by John Hughes

Python Function Scope

In python, variables are visible in the *scope* they are defined in, and all scopes inside.

The scope of the jupyter notebook is the **global scope**.

Functions can use variables defined in the global scope.

```
c = 1
def f(a, b):
    return a + b + c, a - b + c
```

f(1,2) returns (4, 0)

Variables defined *locally* will shadow globals.

```
c = 1
def f(a, b):
    c = 2
    return a + b + c, a - b + c
```

f(1,2) returns (5, 1)

Python Function Scope

Variables defined *locally* are not available outside.

```
def f(a, b):
    return a + b, a - b
f(1,2)
print(a)
```

Output:

```
NameError: name 'a' is not defined
```

Variables defined in loops (including loop index) are available after.

```
for m in range(2):
    print(m)
print(m)
```

gives us:

```
0
1
1
```

First Class Functions

Functions can be assigned to variables: `hypot = lambda x, y : sqrt(x*x + y*y)`.

Functions can also be passed to functions and returned from functions.

Functions passed:

```
def mapit(aseq, func):
    return [func(e) for e in aseq]
mapit(range(3), lambda x : x*x) # [0, 1, 4]
```

map and reduce are famous built-in functions.

Functions returned:

```
def soa(f): # sum anything
    def h(x, y):
        return f(x) + f(y)
    return h
sos = soa(lambda x: x*x)
sos(3, 4) # returns 25 like before
```

It is now generally accepted that modular design is the key to successful programming, and recent languages such as MODULA-II [6] and Ada [5] include features specifically designed to help improve modularity. However, there is a very important point that is often missed. When writing a modular program to solve a problem, one first divides the problem into subproblems, then solves the subproblems, and finally combines the solutions. The ways in which one can divide up the original problem depend directly on the ways in which one can glue solutions together. Therefore, to increase one's ability to modularize a problem conceptually, one must provide new kinds of glue in the programming language. Complicated scope rules and provision for separate compilation help only with clerical details — they can never make a great contribution to modularization.

The power of functional programming comes from better modularization

From: Why Functional Programming Matters, by John Hughes

Pure functions

- **return the same values for the same arguments**, thus being like mathematical functions,
- **no side effects**, by which some state is changed during the execution of the function, and this might not be repeatable, and
- **referential transparenct**, by which one may replace the function call by the resultant return values in the code.

Modularization using higher order functions

- A higher order function is one which takes another function as an argument
- By modularizing a simple function such as sum as a combination of a higher order function (foldr) and some simple arguments including the addition operator, we find that this structure generalizes. We can modularize product as the same foldr plus the multiplication operator
- Indeed list construction, appending, and even applying a given function such as doubling can be written in the same way. By composing a simple function with another for list construction, and by returning a higher order composition from this, we can construct map, a function that applies to every element of a list
- This then generalizes to any data structure, create a foldr and map for it, and go.

Map and Foldr(reduce) plus pure functions

- Many problems can be expressed in this map paradigm where we map a list to another list by some pure function
- Then we reduce or foldr the new list to a result, again via a pure function
- We can do this over huge bits of data by splitting them up into multiple lists over multiple machines
- If a machine fails, the purity comes to our rescue. We just re-run that part of the computation with our pure function, and re-combine it in

MapReduce: Simplified Data Processing on Large Clusters

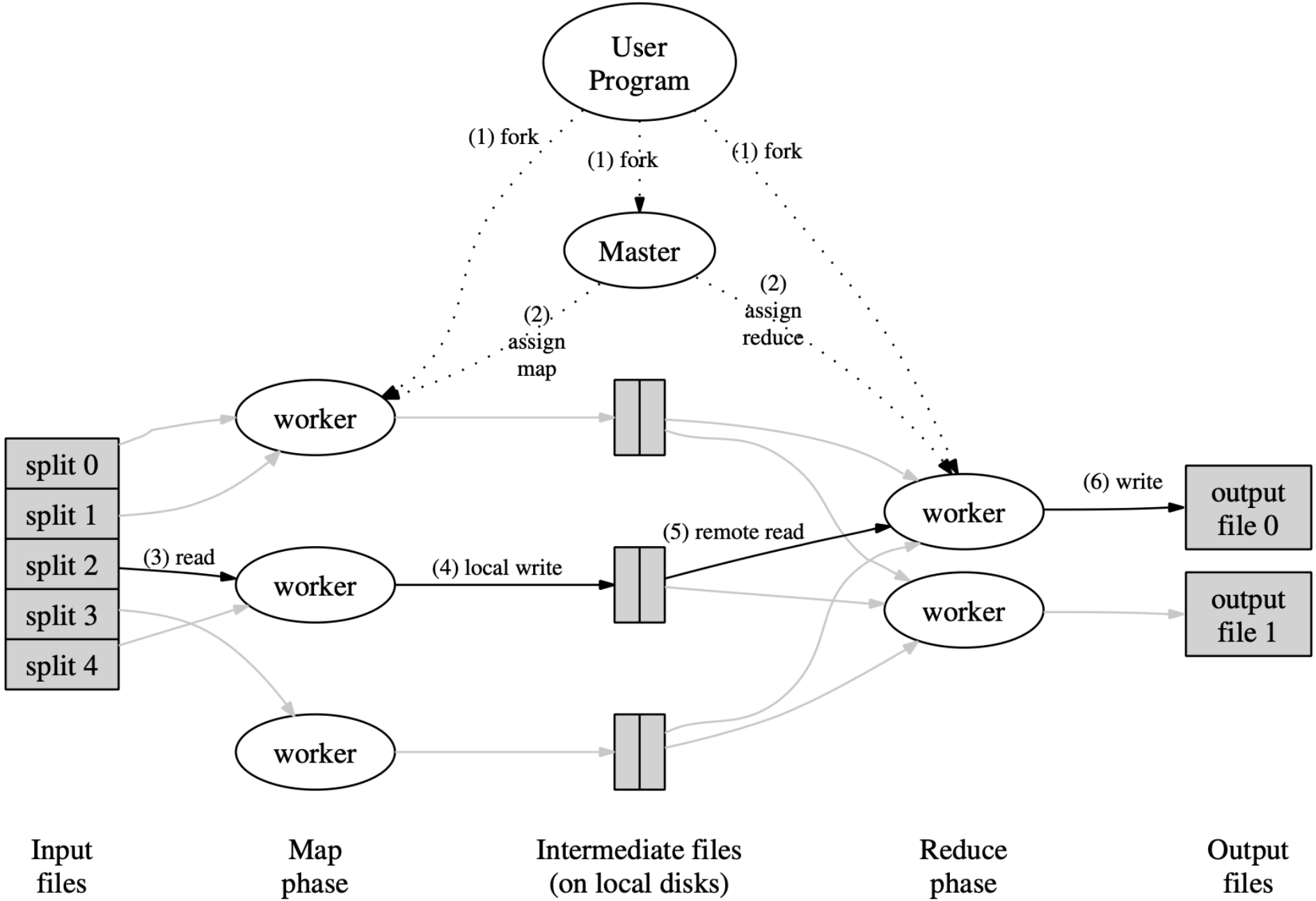
Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

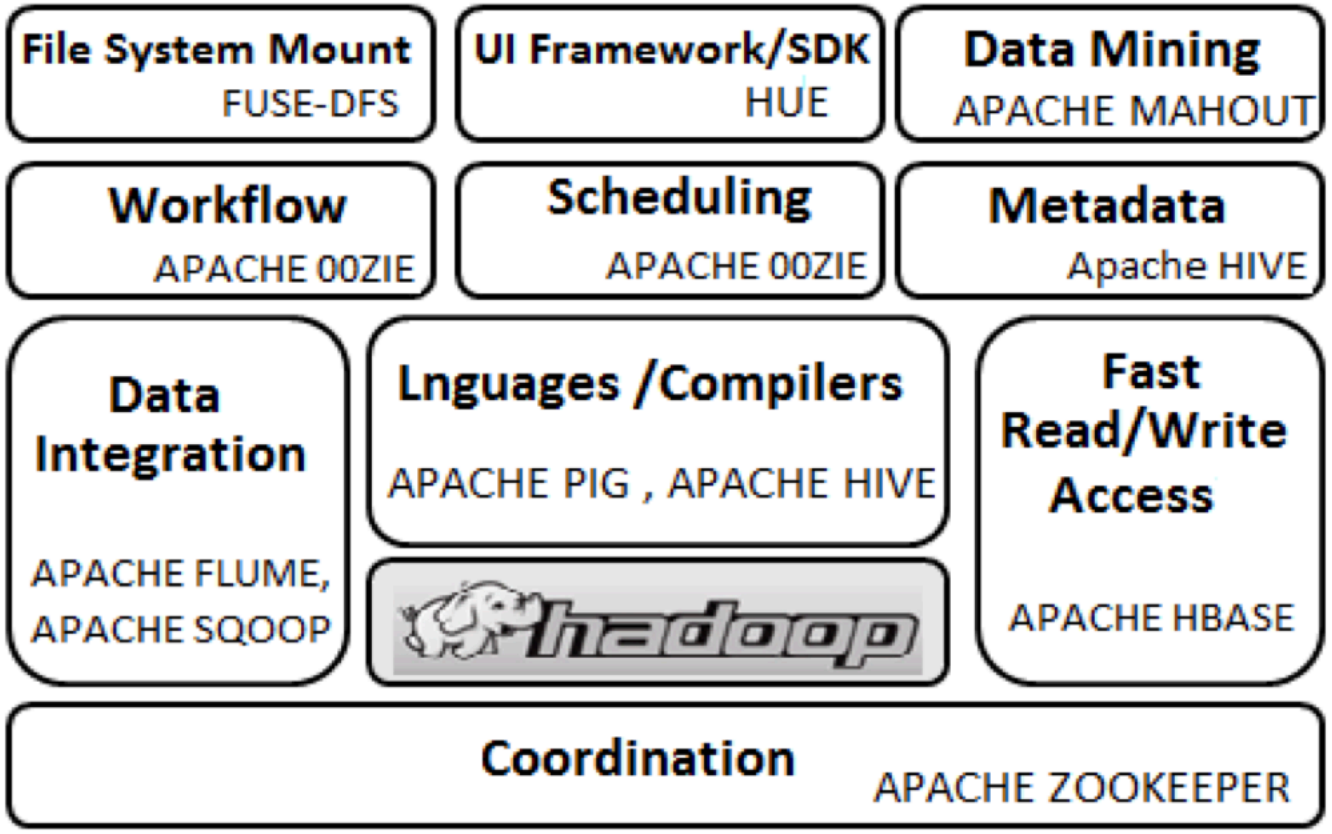
Google, Inc.

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

Hadoop



Hadoop was created as an open-source map-reduce by yahoo. Over time an ecosystem developed around it.

Hadoop is a 200 billion market in 2023!

Closures: capturing state

NOT Pure functions

Sometimes we want to capture some state:

```
def soaplusbias(f, bias): # sum anything
    def h(x, y):
        return f(x) + f(y) + bias
    return h
sosplusbias = soaplusbias(lambda x: x*x, 5)
sosplusbias(3, 4)
```

Returns 30

The last line is indentical to before, but we have additionally captured a `bias` from the enclosing scope. The bias is captured when we define `sosplusbias`, but gets used when we call it. This is called a **closure**. It is useful at all sorts of places where state must be captured and used later, as in deep learning callbacks, GUIs, etc

Decorators

Decorators use the @ syntax and are a shortcut for a function wrapping another.

```
def factorial(n):
    return n*factorial(n-1)

def check_posint(f):
    def checker(n):
        if n > 0:
            return f(n)
        elif n == 0:
            return 1
        else:
            raise ValueError("Not a positive int")
    return checker

factorial = check_posint(factorial)
print(factorial(4)) # returns 24
print(factorial(-1)) # raises a ValueError
```

```
def check_posint(f):
    def checker(n):
        if n > 0:
            return f(n)
        elif n == 0:
            return 1
        else:
            raise ValueError("Not a positive int")
    return checker

@check_posint
def factorial(n):
    return n*factorial(n-1)

print(factorial(4)) # returns 24
print(factorial(-1)) # raises a ValueError
```

Decorators as business value creator

- Decorators exist for just about any activity. You can run functions on specific machines using metaflow/prefect/dagster.
- You can get system observability with tools like datadog
- You can cache your intermediate function results on disk or in memory
- You can create logs to send to log-files for data analysis
- I use it to track my machine learning runs for clients.

- Anytime you want to wrap a function with another function to do “orthogonal” work, use a decorator.

Pseudo-code is an abstraction for real code. Lets see some real code!

Lets see even more python

https://github.com/hult-cm3-rahul/LearningPython