Functional Programming

Functional languages are **declarative** which means they are concerned with *what* needs to be performed as opposed to *how* it should be performed as is the case with procedural imperative languages.

Functional languages rely on **recursion** (the function calling itself) and not iteration.

Functional programs are shorter than codes written in procedural languages. This results in code that is likely to contain fewer errors because there are fewer lines of source code and opportunities to introduce errors.

Properties of Functional Programs

Immutability

Mutable objects can have their values changed.

Immutable objects cannot have their values changed. More strictly when changing the value of an object, we copy to another location in memory and reference that value.

In Python variables are immutable whereas lists are mutable. This can be demonstrated using the `id` function which returns the unique ID of an object. For an immutable object a reference is made to a copy of the object and is shown by different IDs.

Mutable	**Immutable**

`> x=[0]`
`> id(x)`
`64531472`
`> x[0]=1`
`> id(x)`
`64531472`
	`> x=0`
	`> id(x)`
	`1428804032`
	`> x=1`
	`> id(x)`
	`1428804048`

Example: Immutability in Haskell

```
> let f x = x * 5
> f 5
25
> let g x = (f x) * 2
> g 2
20
```

Redefine f

```
> let f x = x * 2
> f 5
10
> g 2
20
```

Our result from *g 2* remains the same because the original definition of f has not been changed (mutated). g was defined in an earlier scope when f had a different value.

No let us do:

```
> let g x = f x * 2
> g 2
8
```

Our result from *g 2* has now changed.

No Side effects

Functional programs have no side effects. Side effects occur when a function changes the values outside its own scope.

Side effects Python	**No side effects: Python**	**# No side effects: Haskell**

`def addOne(x):`
`  x[0]=x[0]+1`
 	
`x=[1]`
`print(x)`
`addOne(x)`
`print(x)`
 	
`Output`
`[1]`
`[2]`
	`def addOne(x):`
	`  x=x+1`
	
	`x=1`
	`print(x)`
	`addOne(x)`
	`print(x)`
	
	`Output`
	`1`
	`1`
		`addOne x = x + 1`
		`addOne 1`
		`2`
		`addOne 1`
		`2`

Statelessness
Given the same input to a function you will always have the same output. The local state is independent of the global state.

Therefore functions can be called in any order without affecting the outcome.

Functional: Stateless	**Procedural: Not stateless**

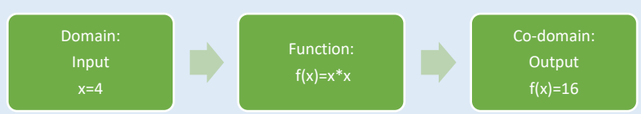
`add x y = x + y`
`add 2 1`
`> 3`
`add 2 1`
`> 3`
	`def add(x):`
	`  global y`
	`  return x+y`
	
	`y=2`
	`print(add(1))`
	`y=3`
	`print(add(1))`
	
	`Output:`
	`3`
	`4`

Functions

Domain - the set of values from which the input is taken

Codomain – the set of values which can be output

Functions map a set of inputs from the domain to a set of output values in the co-domain.



Function Type - Functions have argument data types and a result data types

`F : A -> B`

`A` is the argument type

`B` is the result type

Example
```
double x = x * 2
double :: integer -> integer
```

First class object
In procedural programming Integers, characters and other data types are examples of first class objects. This means they can be:

assigned to variables	`X=5`
passed as arguments to other functions	`Subroutine double(x)` `  return x = x * 2`
returned from other functions	`fx = double(2)`
stored in data structures	`b = [1,2,3,4]`

Functions as First class objects
In functional programming functions can be first class objects.

assigned to variables	

passed as arguments to other functions	`double x = x * 2`
	`sum x y = x + y`
	`sum 2 double 3`

returned from other functions	`add x y = x + y` `addThree = add 3`
stored in data structures	`sum x y = x + y` `a = [sum,1,2,3,4]`

Function application

Function applied to its arguments

Example
```
sum x y  = x y
sum 3 4
```

The type of the function is:
`Add: integer x integer -> integer`
Where `integer x integer` is the Cartesian product

Partial function application

A function that takes multiple parameters is a sequence of single parameter functions.

Example:
```
add :: integer -> integer -> integer
```
is equivalent to
```
add :: integer -> (integer -> integer)
add x y = x + y

add 3 4
```
is equivalent to
```
(add 3) 4
```

Partial function application occurs when a function is applied to fewer than the total number of parameters.

Example:
```
add :: integer -> integer -> integer
add x y = x + y
add3 = add 3
add3 4
7
```

This function is applied to an integer that returns a function that is applied to another integer and returns the sum of the two integers.

Composition of functions
Functions can be combined to create a new function

Example:
```
f x = x * 5
g x = 3 + x^2
f (g 4)
95
```

`g` is applied to the argument `x` first and the `f` is applied to the result of `g`

`f o g x` and `f(g(x))` are equivalent

Codomain type of `g` must be the same as the domain type of `f`

Example:
```
h: float -> integer
k: boolean -> float
```

Thus:
```
h o k: boolean -> float
```
`g o f` is invalid

Higher order functions

Higher-order functions are function that can take a function as an argument and/or return a function as a result.

`Map` works recursively by applying a function to each element of a list and returns the results in a list. It works by applying the function to the head of the list and makes recursive calls to each element in the tail until the list is empty.

Example: Multiply all values in a list by 2
```
map (x2) [1,2,3,4,5] → [2,4,5,6,10]
```

Example: Square all values in a list
```
square x = x * x
map square [1,2,3,4,5] → [1,4,9,16,25]
```

`Filter` applies a function to select items from a list.

Example: select values greater than2 from a list
```
filter (>2) [1,2,3,4,5] → [3,4,5]
```

`Fold` reduces a list to single value using recursion.

Example: Add the numbers in a list
```
fold (+) 6 [1,2,3,4,5] → 21
```

List processing

Returns the first element in a list	`head [3,6,9,1,2]` → 3 `head ["The","cat"]` → The
Returns all but the first element in a list	`tail[3,6,9,1,2]` → `[6,9,1,2]`
Return the length of a list	`length[3,6,9,1,2]` → 5
Set up an empty list	`xs = []`
Check to see if a list is empty	`null xs` → `True`
Append to a list	`[1,2,3,4] ++ [5]` → `[1,2,3,4,5]`
Prepend to a list	`[1] ++ [2,3,4,5] or 1:[2,3,4,5]` → `[1,2,3,4,5]`

Big Data

Big data are voluminous and are often unstructured and come in huge variety of forms making storage in conventional relational databases very difficult or impossible. Multiple machines are required to store, analyse and process the data so distributed processing is important.

Big data have 3 characteristics:

Volume – Huge quantities of data

Velocity – Data are continuously being added to an already huge dataset so greater spare and future capacity is needed

Variety – There is a high range of data types so that data are unstructured

Fact based model for representing data

- Facts are recorded with timestamps
- The are immutable, they cannot be deleted
- Data stored as atomic facts
- Immutable and always true using timestamps
- Each fact uniquely identifiable
- Raw – The rawer the data the better, because one can ask more questions.
- Immutable – Cannot change the data. e.g a record of someone's address does not change.
- Perpetuity – Data will always be true. Although some one moved to a new address, it will always be true that that person lived at another address at a given time.

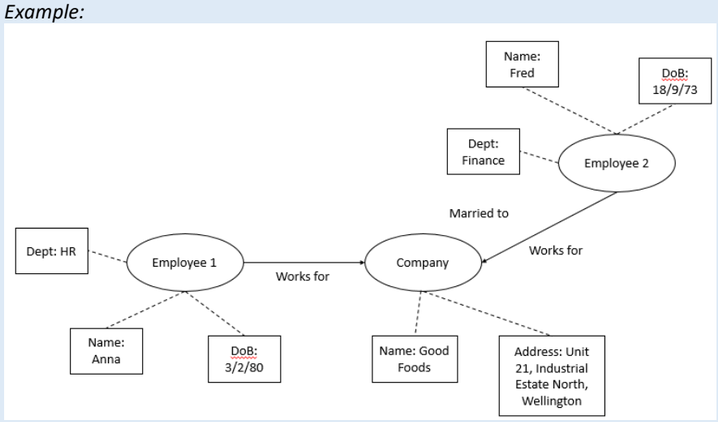
Example: Friends on Social Media

Friends change list	Current friends list	Current friends count
Add James 3/2 Add Bart 9/10 Add Reggie 17/11 Add Grace 18/11 Remove Reggie 12/12 Add Oliver 1/3 Remove Grace 17/12	James Bart Oliver	3

Graph Schemas

Graph schemas are used to represent huge datasets of connected data where nodes (verticies) are the entities and the edges are the relations between facts. Nodes can have properties.

Example:



Why is functional programming good for processing big data?

Because the quantity of data are so large, the data need to be processed in parallel (at the same time) over multiple machines. This means that programs have to be designed with concurrency in mind that is they need to be thread-safe. This is where functional programming comes in.

Functional programming is good from parallel processing because it uses **immutable** data structures, has **statelessness,** there are no **side effects** and uses **higher-order functions**.

Immutable objects are considered more thread safe than mutable objects. That is processing on one thread will not interfere or be interfered by the processing on another thread. This is because for immutable objects there is no danger that other parallel threads will unintentionally change the values of the objects.