FP FOR BEGINNERS

WHAT IS FUNCTIONAL PROGRAMMING?

- Imperative (C, COBOL, Fortran, Pascal)
  - Object-Oriented (Java, C++, Python, most everything)
- Declarative (SQL, Terraform)
  - Logical (Prolog)
  - Reactive (Elm, Svelte, RxJS)
  - Functional (Haskell/Scala, Lisp, Rust-sorta)

KEY POINTS

MATHEMATICAL RIGOR

- Functions are defined in terms of their arguments (domain) and return values (range)
  - Lisp distinguished this by having "Functions"
  - ...and "pseudo-functions"

MATHEMATICAL RIGOR

- Many ideas and idioms grew out of abstract algebra or Category Theory
  - "I have an operation on a type that is associative, commutative, and has an identity element"
  - Examples: Addition, Multiplication, String Concatenation, Appending vectors, merging hash maps or sets...
  - "I want to traverse data structures in a way that only mutates the values, not the structure"
  - Examples: Map, Filter, Reduce/Fold, Zip, etc

FP STYLE

- Product types
  - Tuples, Records, Structs
  - A Person has a Name, Age, and Address, and Customer ID
  - An Order has a Customer ID, List of Product IDs, A unique Order ID, Final Total, and shipping state.

FP STYLE

- Sum Types
  - Enums, Variants
  - A Process can be Stopped, Running, Blocked, or Stopped
  - Your Pasta can be Spaghetti, Penne, or Macaroni
  - The Sauce could be Marinara, Alfredo, Pesto, or Garlic Butter

OMG Stop, just show me code already

IMMUTABLE BY DEFAULT

```python
from typing import Final

A: Final[int] = 5

# Python will run this, but mypy will complain
A = 6
```

SUM TYPES

```
from enum import Enum

class Sauce(Enum):
    Marinara = "Marinara"
    Alfredo = "Alfredo"
    Pesto = "Pesto"
    GarlicAndOil= "Garlic and Oil"

class Pasta(Enum):
    Spaghetti = "Spaghetti"
    Penne = "Penne"
    Bowtie = "Bowtie"
```

PRODUCT TYPES

```
from typing import NamedTuple

class Dish(NamedTuple):
    pasta: Pasta
    sauce: Sauce

my_favorite_dish = Dish(pasta=Pasta.Spaghetti, sauce=Sauce.Gar

print(my_favorite_dish)
```

PATTERN MATCHING

```
match my_favorite_dish.sauce:
    case Sauce.Marinara:
        print("Pairs well with red wine.")
    case Sauce.Alfredo:
        print("Pairs wonderfully with Salmon or Shrimp.")
    case Sauce.Pesto:
        print("Pesto is great on a summer day.")
    case Sauce.GarlicAndOil:
        print("Weeknight meal of champions!")
```

PATTERN MATCHING, EXTENDED:

```
match my_favorite_dish:
    case Dish(pasta=Pasta.Spaghetti, sauce=Sauce.GarlicAndOil)
        print("Yum! My favorite dish!")
    case Dish(pasta=Pasta.Penne, sauce=Sauce.Marinara):
        print("Penne with Marinara is good too!")
    case _:
        print("I like all pasta dishes!")
```

HIGHER ORDER FUNCTIONS

Let's detour into Lisp. Say we have a list of integers:

```
(define my-list '(1 2 3 4 5))
```

HIGHER ORDER FUNCTIONS

Now, we want to add one to each element in the list.

We can do this with a function:

```
(define my-list '(1 2 3 4 5))

(define (add-one xs)
  (if (null? xs)
      '()
      (cons (+ 1 (car xs)) (add-one (cdr xs)))))

(add-one my-list)
;;; prints '(2 3 4 5 6)
```

HIGHER ORDER FUNCTIONS

That's great! Jira ticket closed. Next, we are told that we need a function to square each number.

```
(define my-list '(1 2 3 4 5))

(define (square xs)
  (if (null? xs)
      '()
      (cons (* (car xs) (car xs)) (add-one (cdr xs)))))

(square my-list)
;;; prints '(1 4 9 16 25)
```

We can imagine that next we'll get a prompt to cube the list, or do some other transform.

So how do we factor out the part we keep reciting?

```
(define (map fn xs)
  (if (null? xs)
      '()
      (cons (fn (car xs)) (map fn (cdr xs)))))
```

```
(define (map fn xs)
  (if (null? xs)
      '()
      (cons (fn (car xs)) (map fn (cdr xs)))))

(define (add-one x) (+ 1 x))
(define (square x) (* x x))
(map add-one my-list)
;;; prints '(2 3 4 5 6)
(map square my-list)
;;; prints '(1 4 9 16 25)
```

```
(map (lambda (x) (+ x 1)) my-list)
;;; prints '(2 3 4 5 6)
(map (lambda (x) (* x x)) my-list)
;;; prints '(1 4 9 16 25)
```

```python
myList: list[int] = [1, 2, 3, 4, 5]

myLambda: Callable[[int], int] = lambda x: x * x
myNewList: list[int] = list(map(myLambda, myList))

# prints [1, 4, 9, 16, 25]
print("Mapped List:")
print(myNewList)
```

HIGHER ORDER FUNCTIONS

Similarly, we can imagine a function that only returns the even numbers in a list.

```
(define (even-list xs)
  (if (null? xs)
      '()
      (if (even? (car xs))
      (cons (car xs) (even-list (cdr xs)))
      (even-list (cdr xs)))))
```

We can also imagine that we'd want to filter only the prime numbers, or the odd numbers, or the numbers that are greater than 10, and we'd rather not re-write each time

```
(define (filter fn xs)
  (if (null? xs)
      '()
      (if (fn (car xs))
      (cons (car xs) (filter fn (cdr xs)))
      (filter fn (cdr xs)))))


(filter (lambda (x) (= 0 (% x 2))) my-list)
(filter (lambda (x) (= 1 (% x 2))) my-list)
(filter (lambda (x) (> 10 x) my-list)
;;; a function that detects primes is
;;; left as an exercise to the reader :)
```

```python
myFilterLambda: Callable[[int], bool] = lambda x:
x % 2 == 0

myFilteredList: list[int] = list(
    filter(myFilterLambda, myList))

# prints [2, 4]
print("Filtered List:")
print(myFilteredList)
```

HIGHER ORDER FUNCTIONS

Finally, we can imagine a function that reduces a collection to a single value.

```
(define (sum xs)
  ;; Locally define a helper function
  ;; to encapsulate the accumulation logic
  (define (go xs acc)
    (if (null? xs)
      acc
      (go (cdr xs) (+ acc (car xs)))))
;; Then call that function, and return its result
(go xs 0))
```

If we're playing at being data scientists, we might also want to square a list of numbers, and then sum them up.

```
(define (sum-of-squares lst)
  ;; Locally define a helper function to encapsulate the accum
  (define (squared-list xs)
    (if (null? xs)
        '()
        (cons (* (car xs) (car xs)) (squared-list (cdr xs)))))
  (define (go xs acc)
    (if (null? xs)
      acc
      (go (cdr xs) (+ acc (car xs)))))
;; Then call that function, and return its result
  (go (squared-list lst) 0))

(sum-of-squares my-list)
;;; prints 55
```

Clearly, there's an easier way.

```
(define (reduce fn acc xs)
  (if (null? xs)
      acc
      (reduce fn (fn acc (car xs)) (cdr xs))))

(reduce + 0 (map (lambda (x) (* x x)) my-list))
;;; prints 55
```

We can even carry this further. Say we want to sum the squares of the even numbers in a list.

```
(reduce + 0
  (map (lambda (x) (* x x))
    (filter (lambda (x) (= 0 (% x 2))) my-list)))
;;; prints 20
```

```python
from functools import reduce

# Reduce functions takes two arguments,
# the current accumulated value and the
# next element in the list. The result
# is assigned to the accumulator for
# the next iteration.
myReduceLambda: Callable[[int, int], int] =
    lambda acc, elem: acc + elem

myReducedValue: int =
    reduce(myReduceLambda, myList)

print("Reduced Value:")
print(myReducedValue)
```

TYPE SAFETY AND NULL

Let's turn to rust, to demonstrate nulls vs optionals.

```
fn fn_1() {
    let maybe_value: Option = Some(42);

    match maybe_value {
        Some(value) => println!("The value is: {}", value),
        None => println!("No value found"),
    }
}
```

Let's see it in use:

```
fn match_get(res: Option) {
    match res {
        Some(value) => println!("The value is: {}", value),
        None => println!("No value found"),
    }
}
fn fn_2() {
    let my_list = vec![1, 2, 3, 4, 5];
    match_get(my_list.get(2).cloned());
    match_get(my_list.get(10).cloned());

    // As opposed to:
    // my_list[10]; // This would panic at runtime
}
```

```python
# Use a function that returns an optional
# instead of an exception.
def safe_divide(numerator: int, denominator: int)
      -> float | None:
  if denominator == 0:
    return None
  return numerator / denominator
```

```python
# Example usage of safe_divide
result: float | None = safe_divide(10, 2)
if result is None:
    print("Division by zero is not allowed.")
    exit(code=0)

print(f"Result of division: {result}")
```

Now, what if we could fail for many reasons?

```
enum FileReadError {
    NotFound,
    PermissionDenied,
    FileAlreadyOpen,
}
```

```rust
fn fn_3() {
    let result_value: Result = Ok(100);

    match result_value {
        Ok(value) => println!("The value is: {}", value),
        Err(FileReadError::NotFound) =>
            println!("File not found"),
        Err(FileReadError::PermissionDenied) =>
            println!("Permission denied"),
        Err(FileReadError::FileAlreadyOpen) =>
            println!("File is already open"),
    }
}
```

Python has a notion of "I return this type or that" which we've seen above. This is very similar to Either in Haskell, or Result in Rust. We'll use this to demonstrate how to return an error that is not an exception, but still gives context on what went wrong.

```python
class FileReadError(Enum):
  FILE_NOT_FOUND = "File not found"
  IO_ERROR = "I/O error occurred"

def read_file_with_union(file_path: str)
    -> str | FileReadError:
  try:
    with open(file_path, 'r') as file:
      return file.read()
  except FileNotFoundError:
    return FileReadError.FILE_NOT_FOUND
  except IOError as e:
    print(f"An error occurred while reading the file: {e}")
    return FileReadError.IO_ERROR
```

MOTIVATION

Slide 27 from a talk from Tim Sweeney, CEO of Epic Games

“The Next Mainstream Programming Language: A Game Developer’s Perspective”

Slide deck:
https://groups.csail.mit.edu/cag/crg/papers/sweeney06

Dynamic Failure in Mainstream Languages

Solved problems:

- Random memory overwrites
- Memory leaks

Solveable:

- Accessing arrays out-of-bounds
- Dereferencing null pointers
- Integer overflow
- Accessing uninitialized variables

50% of the bugs in Unreal can be traced to these problems!

Null pinters, and array-out-of-bounds are solved by Maybe/Optional and Either/Result. Failure is explicit, and the type system guides you through handling both cases.

Int overflow can be solved by having an unbounded "Integer" type

Uninitialized variables are ususally caused by declaring a variable that can have different values depending on the result of a conditional.

You can't make this change, as it's language-level, but expression-based languages make this a lot easier to avoid.