Functional Programming

Announcements

What's Next?

New Programming Language Ideas

- Algebraic data types and pattern matching (Rust, Swift, TypeScript, etc.)
- Working effectively with immutable data (Rust, Erlang, Scala, etc.)
- Value-based error handling (Rust, Go, Elm, etc.)

Interpreters

- Implementing the rules for how programs run
- Controlling how programs run

Applications

- Program design and web technologies
- AI-assisted software development

Functional Programming

Perspective on Programming

Imperative: Programs are instructions for updating memory

Object-oriented: Programs define objects that behave like entities they represent

Functional: Programs are data definitions and pure functions that generate data

These aren't mutually exclusive, and ideas from each perspective can be combined

The Gleam Language

Some characteristics of Gleam that make it a functional language:

- Program behavior is just functions, and function bodies are expressions
- Functions are first-class values that can be passed as arguments and returned
- Anonymous functions are convenient to create and used liberally
- All values are immutable (and no for/while loops that update variables)

Why learn Gleam in particular?

- It's small & simple: only functions, names, and one form of conditional (case)
- A nice language for learning about algebraic data types and pattern matching
- Great preparation for learning Rust, Swift, and Typescript (which are larger)

Gleam Functions

Gleam Programs

Gleam programs are organized into modules (files). For a while, we'll just write one module.

There must be a main function that is accessible to users of the module (marked as **pub**):

```
pub fn main() {
  echo 61
}
```

echo displays any value, similar to Python's print function

You can also define more functions and call them.

https://playground.gleam.run/

(Demo)

Expressions

An expression is evaluated in an environment (that gives names meaning) to produce a value.

The body of Gleam functions are all expressions, which can be:

- Literal expressions: `2` `3.3` `True` `"Hello"`
- Call expressions: `f(2, 3)`
- Let expressions: `let x = 4`
- Echo expressions: `echo x`
- Case expressions:

```
case x > 3 {
    True -> "big"
    False -> "not big"
}
```

```
case x {
    3 -> "three"
    4 -> "four"
    _ -> "not three or four"
}
```

(Demo)

Defining Functions

No **return** in Gleam; the value of a call expression is the value of its body expression

```python
def sum_squares(x, y):
    return x * x + y * y
```

```
fn sum_squares(x, y) {
  x * x + y * y
}
```

Instead of multiple return statements, Gleam returns the value of the case expression

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 2) + fib(n - 1)
```

```
fn fib(n) {
  case n {
    0 -> 0
    1 -> 1
    _ -> fib(n - 2) + fib(n - 1)
  }
}
```

Guards in Case Expressions

There is **if** in Gleam! It introduces a guard on a pattern in a **case** expression

```gleam
pub fn describe(n: Int) -> String {
  case n {
    _ if n < 0 -> "Negative"
    0 -> "None"
    1 -> "One"
    2 | 3 | 4 -> "Few"
    _ -> "Many"
  }
}
```

```gleam
pub fn sign(n: Int) -> Int {
  case n {
    _ if n > 0 -> 1
    _ if n < 0 -> -1
    _ -> 0
  }
}
```

No reference of n

Nil is "a value that is returned by functions that have nothing else to return, as all functions must return something."

```gleam
pub fn sign(n: Int) -> Int {
  case Nil {
    _ if n > 0 -> 1
    _ if n < 0 -> -1
    _ -> 0
  }
}
```

Higher-Order Functions

Function Types and Anonymous Functions

An anonymous function can be created with fn: `fn(x, y) { x + y }`

A function type can be declared with fn: `fn(Int, Int) -> Int`

These are used together to create higher-order functions

A function type for a one-argument function of integers

```
fn make_adder(n: Int) -> fn(Int) -> Int {
  fn(k) { n + k }
}
```

Calling make_adder returns a function, which can be called to add 3 and 4

```
make_adder(3)(4)
```

Calling combine on a function that multiplies

```
combine(2, 3, 4, fn (x, y) { x * y })
```

```
fn combine(x: Int, y: Int, z: Int,
           f: fn(Int, Int) -> Int) -> Int {
  f(f(x, y), z)
}
```

A function type for a two-argument function of integers

Assignment

Gleam Let

Assign a value to a local name with let:

```
fn sum_squares(x, y) {
  let x2 = x * x
  let y2 = y * y
  x2 + y2
}
```

Is equivalent to

```
fn sum_squares(x, y) {
  fn (x2) {
    fn (y2) {
      x2 + y2
    }(y * y)
  }(x * x)
}
```

Each let introduces a new scope (equivalent to a new local frame)

```
>>> x = 3
>>> f = lambda: x
>>> x = 4
>>> f()
4
```

```
pub fn main() {
  let x = 3
  let f = fn () {x}
  let x = 4
  echo f()
}
```

```
src/main.gleam:5
3
```

```
warning: Unused variable
   ┌─ /src/main.gleam:4:7
   │
4  │   let x = 4
   │       ^ This variable is never used

Hint: You can ignore it with an underscore: `_x`.
```

Blocks

Blocks

Blocks can contain multiple lines, and they evaluate to the value of their last expression

A let within a block does not affect expressions outside the block

Blocks are used to change the order of evaluation in arithmetic (instead of parentheses)

```
/// Compute a*x*x + b*x + c
pub fn quadratic(x: Int, a: Int, b: Int, c: Int) -> Int {
  let ax2 = {
    let x2 = x * x
    a * x2
  }

  // ax2 is in scope here, but x2 is not
  ax2 + b * x + c
}


let celsius = { fahrenheit - 32 } * 5 / 9
echo celsius
```

Iteration

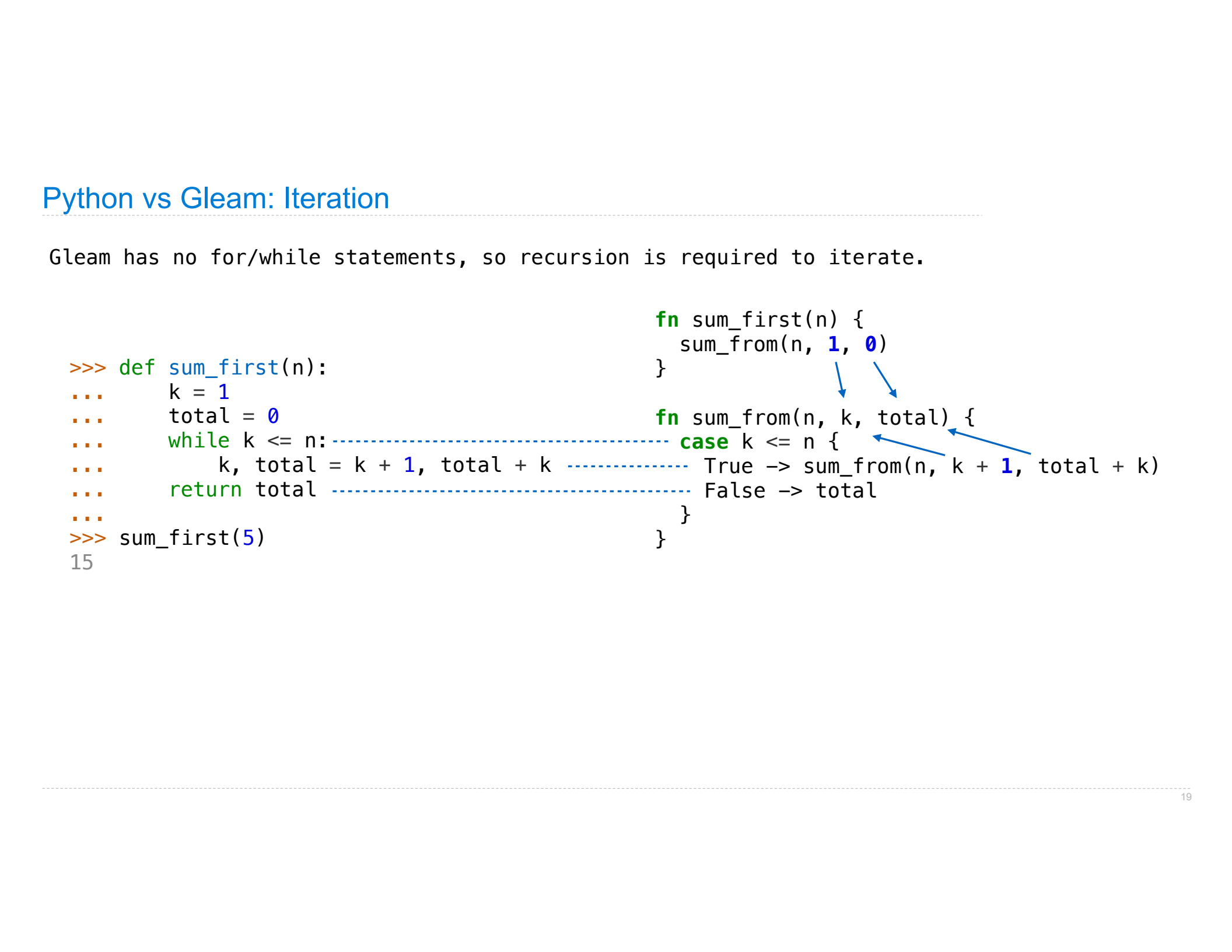
Python vs Gleam: Iteration

Gleam has no for/while statements, so recursion is required to iterate.

```
>>> def sum_first(n):
...     k = 1
...     total = 0
...     while k <= n:
...         k, total = k + 1, total + k
...     return total
...
>>> sum_first(5)
15
```

```
fn sum_first(n) {
  sum_from(n, 1, 0)
}

fn sum_from(n, k, total) {
  case k <= n {
    True -> sum_from(n, k + 1, total + k)
    False -> total
  }
}
```



Evaluation by Substitution

With no loops or mutation, call expressions can be evaluated by *substitution*:

To apply a function to arguments, evaluate the body of the function with each formal parameter replaced by the corresponding argument.

```
fn sum_first(n) {
  case n {
    0 -> 0
    _ -> n + sum_first(n-1)
  }
}
```

```
sum_first(5)
```

```
5 + sum_first(4)
```

```
5 + 4 + sum_first(3)
```

```
5 + 4 + 3 + sum_first(2)
```

```
5 + 4 + 3 + 2 + sum_first(1)
```

```
5 + 4 + 3 + 2 + 1 + sum_first(0)
```

```
5 + 4 + 3 + 2 + 1 + 0
```

Tail Recursion

Comparing Two Processes

Some recursive processes grow, while others have a fixed size.

```
fn sum_first(n) {
  case n {
    0 -> 0
    _ -> n + sum_first(n-1)
  }
}
```

```
sum_first(5)
5 + sum_first(4)
5 + 4 + sum_first(3)
5 + 4 + 3 + sum_first(2)
5 + 4 + 3 + 2 + sum_first(1)
5 + 4 + 3 + 2 + 1 + sum_first(0)
5 + 4 + 3 + 2 + 1 + 0
```

```
fn sum_first(5) {
  sum_from(5, 1, 0)
}

fn sum_from(5, 1, total) {
  case 1 > 5 {
    True -> total
    False -> sum_from(5, 1 + 1, total + 1)
  }
}
```

```
sum_first(5)
sum_from(5, 1, 0)
sum_from(5, 2, 1)
sum_from(5, 3, 3)
sum_from(5, 4, 6)
sum_from(5, 5, 10)
sum_from(5, 6, 15)
15
```

Tail Calls

A language that executes an iterative process as effeciently as a while statement, even if the iterative process is described by a recursive procedure, is called *tail-recursive*.

```
fn sum_from(n, k, total) {
  case k > n {
    True -> total
    False -> sum_from(n, k + 1, total + k)
  }
}
```

Should use resources like

```
def sum_from(n, k, total):
    while k <= n:
        k, total = k + 1, total + k
    return total
```

This works in Gleam (and tail-recursive languages in general) if the recursive call appears in a *tail context*:

- The body of a function (after any **let**)
- A result of a **case** in a tail context

Non-Tail Calls

A recursive call that's part of a larger expression is **not** in a tail context.

```
fn sum_from(n, k, total) {
  case k > n {
    True -> total
    False -> sum_from(n, k + 1, total + k)
  }
}
```

This works in Gleam (and tail-recursive languages in general) if the recursive call appears in a *tail context*:

- The body of a function (after any **let**)
- A result of a **case** in a tail context

```
fn sum_first(n) {
  case n {
    0 -> 0
    _ -> n + sum_first(n-1)
  }
}
```

Not a tail context

When the recursive call is in a tail context, calling the function is fast and uses space just like a while loop.

Compiling Gleam

Gleam is compiled (checked and translated) into either Javascript or Erlang.

When compiling to Javascript, which is not tail-recursive, it generates while loops for recursive tail calls.

```
fn sum_from(n, k, total) {
  case k > n {
    True -> total
    False -> sum_from(n, k + 1, total + k)
  }
}
```

```
function sum_from(loop$n, loop$k, loop$total) {
  while (true) {
    let n = loop$n;
    let k = loop$k;
    let total = loop$total;
    let $ = k > n;
    if ($) {
      return total;
    } else {
      loop$n = n;
      loop$k = k + 1;
      loop$total = total + k;
    }
  }
}
```

Demo: https://playground.gleam.run/

Example: Tail-Recursive Fib

Try to implement fib as a tail-recursive function.

```
fn fib(n) {
  case n {
    0 | 1 -> n
    _ -> fib(n-2) + fib(n-1)
  }
}
```

```
fn fib(n) {
  fib_iter(1, 0, n)
}

fn fib_iter(prev, curr, steps) {
  case steps {
    0 -> curr
    _ -> fib_iter(curr, prev + curr, steps - 1)
  }
}
```

```
def fib(n):
    prev, curr = 1, 0
    while n > 0:
        prev, curr, n = curr, prev + curr, n - 1
    return curr
```

https://playground.gleam.run/

(Careful: making an infinite loop will crash the playground)

Generic Functions

Generic Functions Take Multiple Types of Values

A lowercase type name is a placeholder for the actual type used to call a function.

```gleam
pub fn main() {
  let add_one = fn(x) { x + 1 }
  let exclaim = fn(x) { x <> "!" }

  echo twice(add_one, 10)
  echo twice(exclaim, "Hello")
}

// The name t refers to the same type multiple times
fn twice(f: fn(t) -> t, x: t) -> t {
  f(f(x))
}
```

```gleam
// Also works:
fn twice(f, x) {
  f(f(x))
}
```

Exercise: Write the generic function `fn k_times(f: fn(t) -> t, x: t, k: Int) -> t`, which calls `f` repeatedly `k` times on `x`.
For example, `k_times(add_one, 10, 3)` evaluates `add_one(add_one(add_one(10)))`

https://playground.gleam.run/