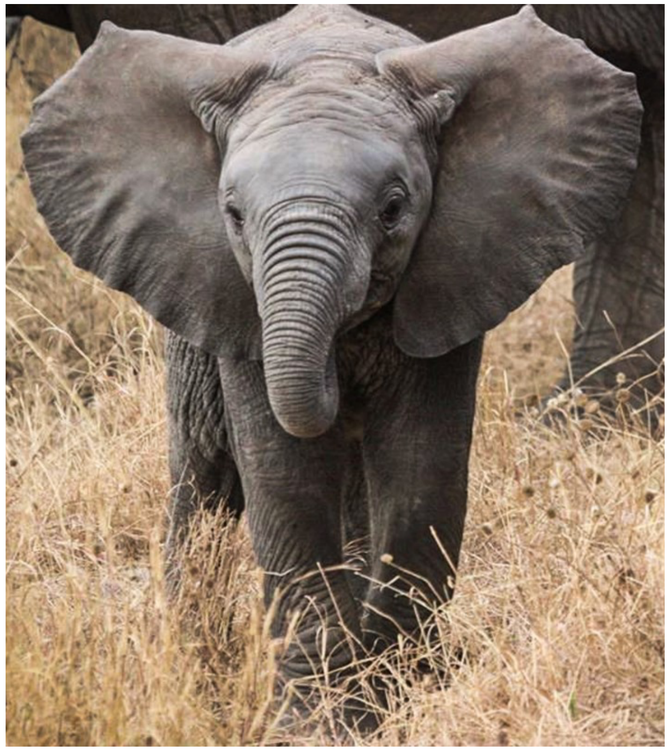
Functional Programming

Functional Programming

Functional Programming

- Programming Paradigms
- Iterators
- Generators
  - Expensive operations, infinite streams
- `lambda`
- `map`, `filter`, `functools.reduce`
- Decorators



Alt Text: a baby elephant with Week 7 vibes.
(Image Source)

Programming Paradigms

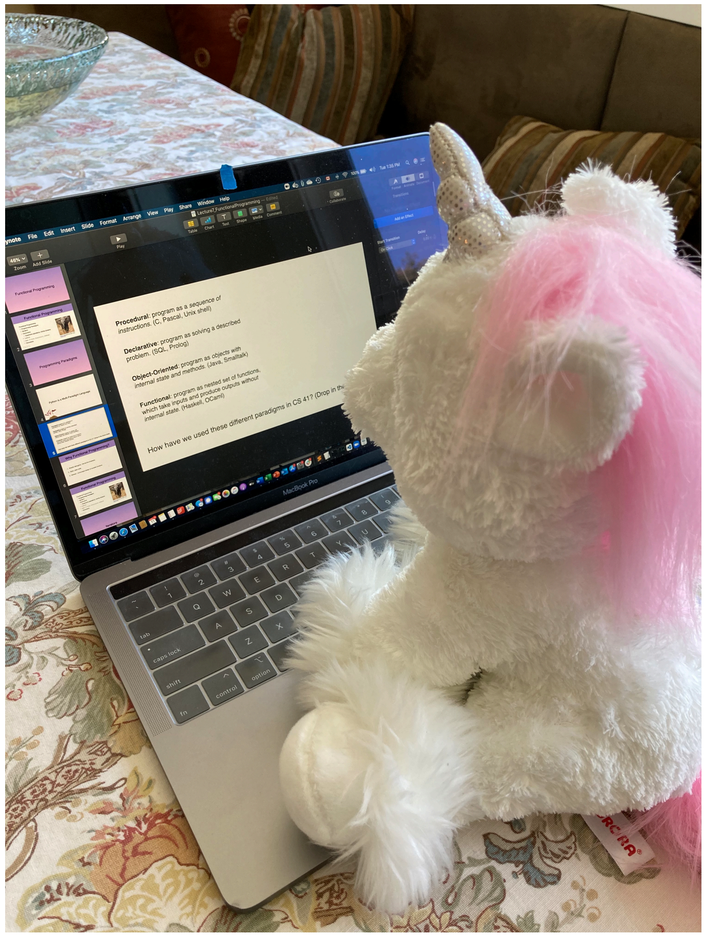
Python is a Multi-Paradigm Language

Procedural: program as a *sequence of instructions*. (C, Pascal, Unix shell)

Declarative: program as solving a described problem. (SQL, Prolog)

Object-Oriented: program as *objects with internal state and methods*. (Java, Smalltalk)

Functional: program as nested set of functions, which take inputs and produce outputs *without internal state*. (Haskell, OCaml)



Alt Text: Unicornelius (pre-kidnapping) thinking this slide looks rather bare without a photo.

How have we used these different paradigms in CS 41? (Drop in the Zoom chat!)

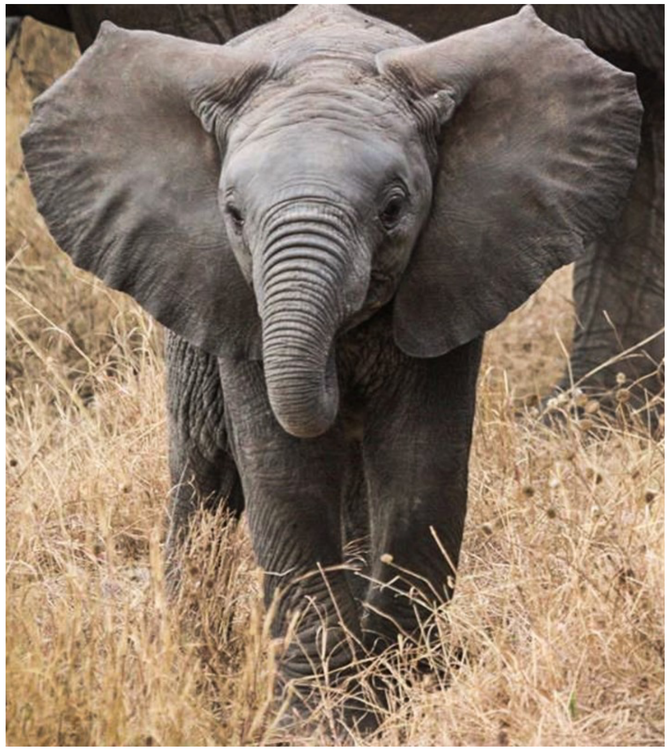
Why Functional Programming?

1. Simplify debugging - line-by-line invariants.
2. Short, clean code.
3. Modular - encourages design of reusable functions.

Functional Programming

Functional Programming

- ~~Programming Paradigms~~
- Iterators
- Generators
  - Expensive operations, infinite streams
- `lambda`
- `map`, `filter`, `functools.reduce`
- Decorators



Alt Text: a baby elephant with Week 7 vibes.
(Image Source)

Iterators

Iterator: a representation of a finite or infinite stream of data.

Iterators: Summary

- Iterators are an abstraction used to represent a stream of data.
- Use `next(iter)` to obtain the next value in the stream.
  - Upon reaching the end of the stream, `next(iter)` raises a StopIteration error.
- The constructor `iter(data_structure)` allows us to build an iterator over a data structure.
  - (Finite) iterators can be read into data structures using standard constructors. (E.g. `list(some_iterator)`)
- Once exhausted, iterators cannot be reused or reset: they must be re-initialized.

```
for data in iterable:
    do_something_to(data)
```

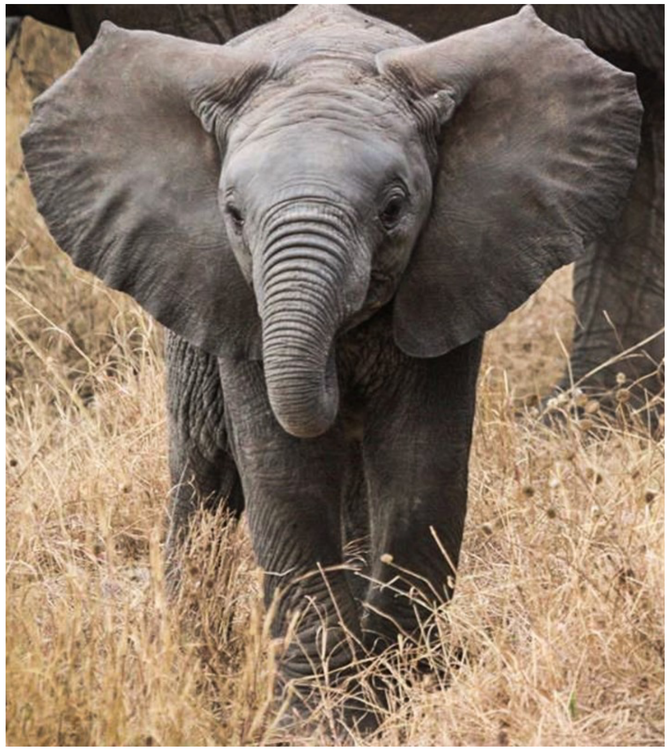
==

```
data_iter = iter(iterable)
while True:
    try:
        data = next(data_iter)
    except StopIteration:
        break
    else:
        do_something_to(data)
```

Functional Programming

Functional Programming

- ~~Programming Paradigms~~
- ~~Iterators~~
- Generators
  - Expensive operations, infinite streams
- `lambda`
- `map`, `filter`, `functools.reduce`
- Decorators



Alt Text: a baby elephant with Week 7 vibes.
(Image Source)

Generators

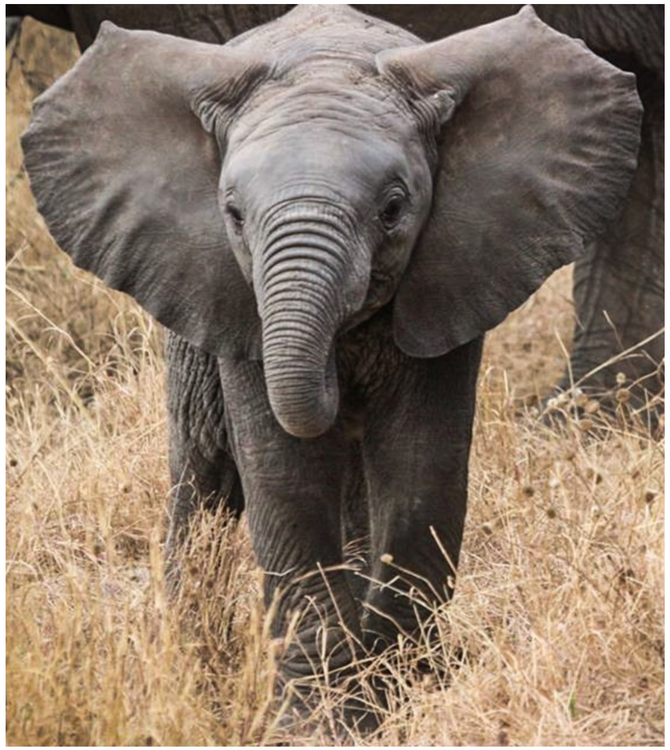
Generators: Summary

- Generators are a special type of iterator: a function that preserves local namespace when suspended. A generator is suspended anytime execution hits the `yield` keyword within it.
- Define a generator using the `yield` keyword within a function. `return`ing from a generator produces a `StopIteration` error.
- Use cases:
  - Compute data on demand - helps us deal with expensive function calls, memory management, or reducing memory buffering.
  - Elegantly deal with infinite streams of data.

Functional Programming

Functional Programming

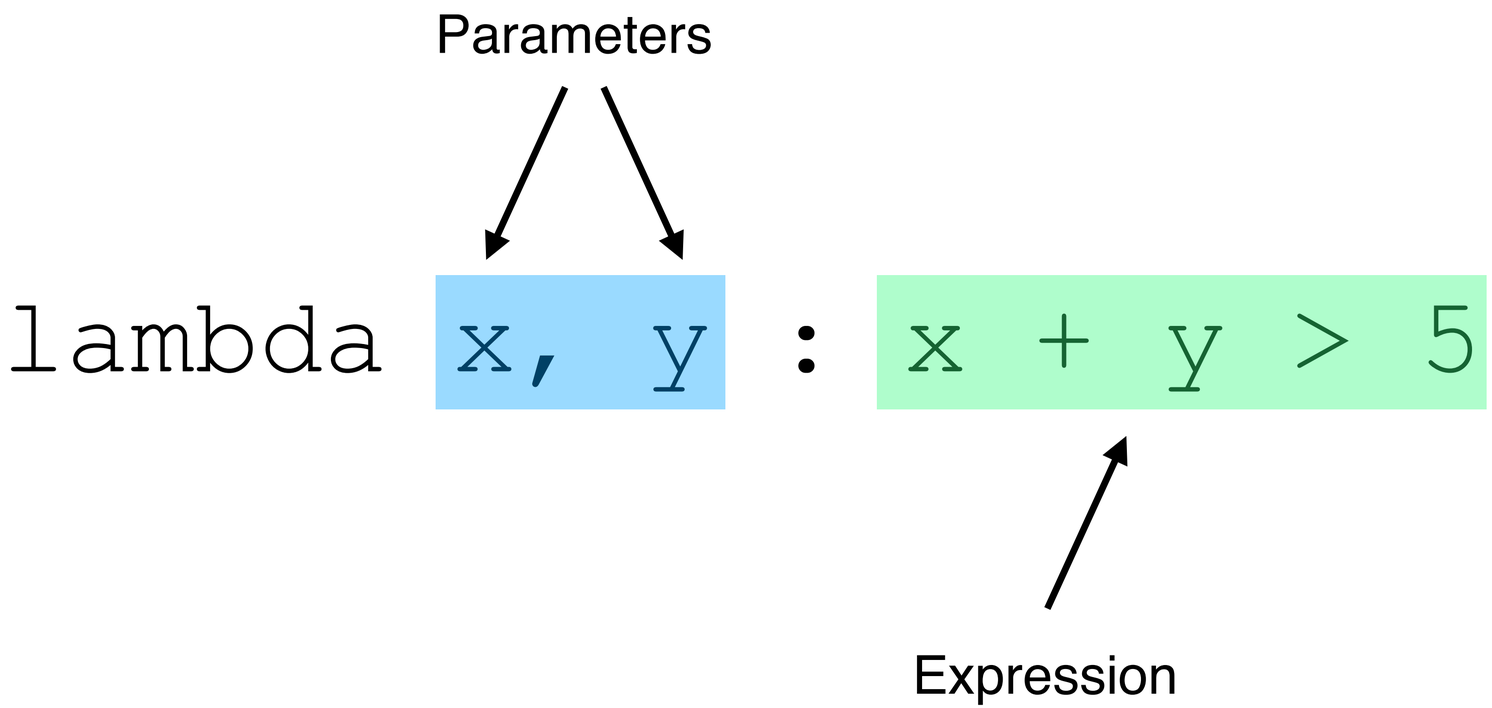
- ~~Programming Paradigms~~
- ~~Iterators~~
- ~~Generators~~
  - ~~Expensive operations, infinite streams~~
- `lambda`
- `map`, `filter`, `functools.reduce`
- Decorators



Alt Text: a baby elephant with Week 7 vibes.
(Image Source)

lambda

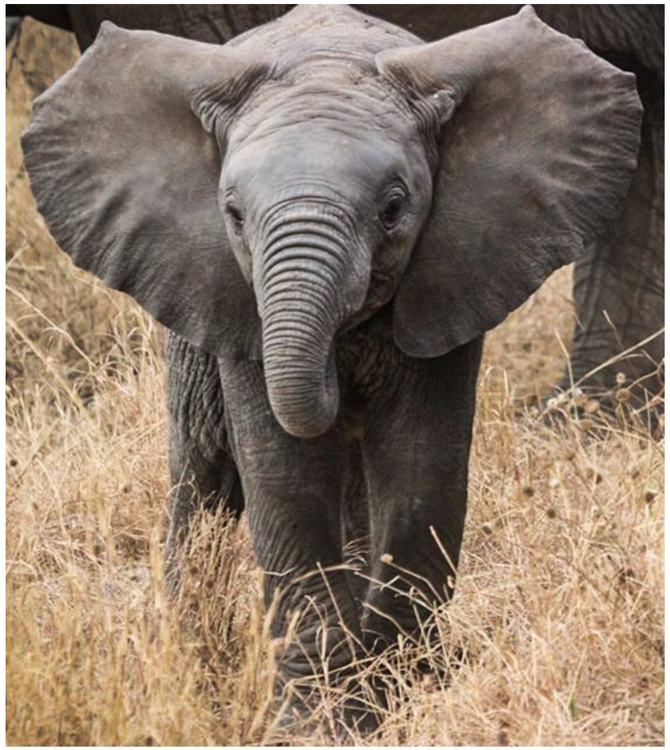
`lambda:` an inline, anonymous function, designed to be used once, then discarded.


Parameters
lambda x, y : x + y > 5
Expression

Functional Programming

Functional Programming

- ~~Programming Paradigms~~
- ~~Iterators~~
- ~~Generators~~
  - ~~Expensive operations, infinite streams~~
- ~~`lambda`~~
- `map`, `filter`, `functools.reduce`
- Decorators



Alt Text: a baby elephant with Week 7 vibes.
(Image Source)

`map`, `filter`, and `functools.reduce`

`map`: returns a new iterable consisting of `fn` applied to each element of `iterable`.

```
map(fn, iterable)
```

```
map(fn, iterable)
```

vs.

```
[fn(x) for x in iterable]
```

`filter`: returns a new iterable consisting of the elements of `iterable` for which `fn(elem)` is `True`.

```
filter(fn, iterable)
```

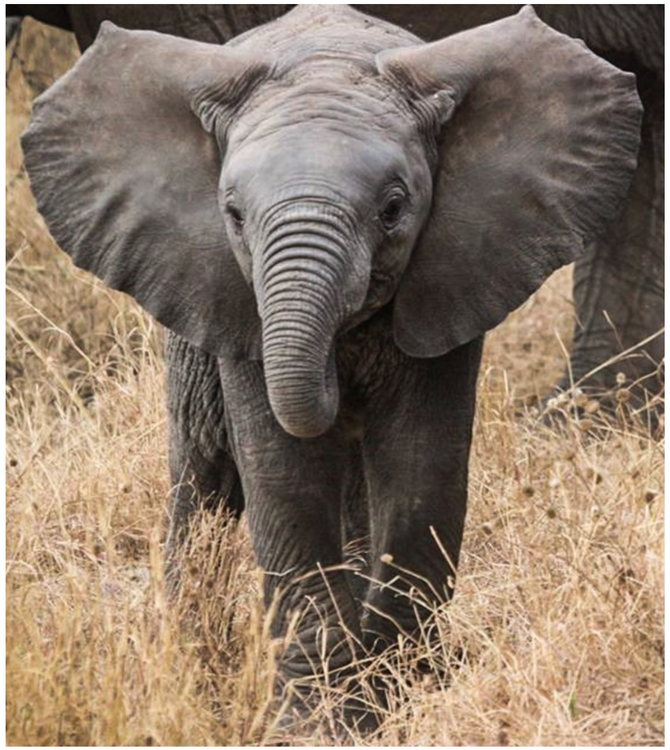
`functools.reduce`: applies `fn` of two arguments cumulatively to items of `iterable`, left to right, to reduce the iterable to a single value.

```
functools.reduce(fn, iterable)
```

Functional Programming

Functional Programming

- ~~Programming Paradigms~~
- ~~Iterators~~
- ~~Generators~~
  - ~~Expensive operations, infinite streams~~
- ~~`lambda`~~
- ~~`map`, `filter`, `functools.reduce`~~
- Decorators



Alt Text: a baby elephant with Week 7 vibes.
(Image Source)

Decorators

Decorator: a function that accepts another function as a parameter, modifies it, then returns the parameter function.

A Familiar Decorator

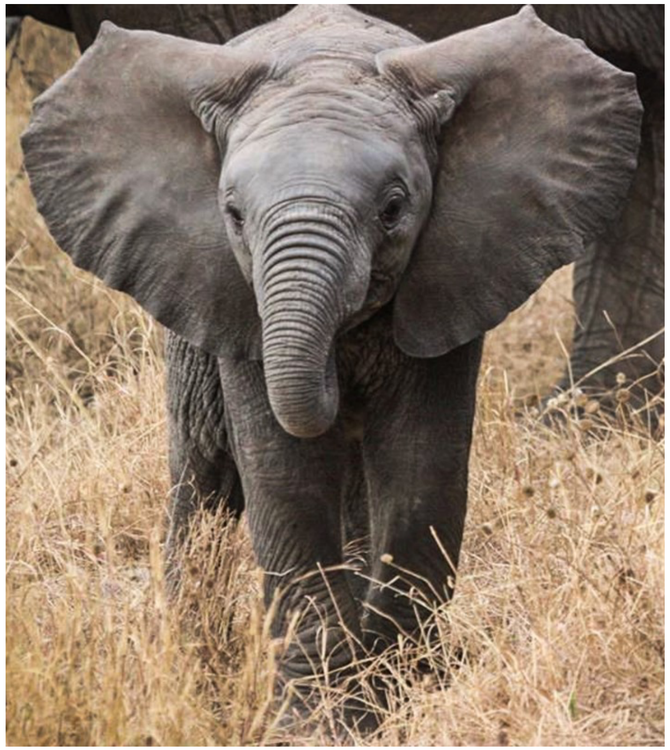
```
@app.route("/")
def main():
    return "Hello, world!"
```

```
def main():
    return "Hello world"
app.add_url_rule("/", "main", main)
```

Functional Programming

Functional Programming

- ~~Programming Paradigms~~
- ~~Iterators~~
- ~~Generators~~
  - ~~Expensive operations, infinite streams~~
- ~~`lambda`~~
- ~~`map`, `filter`, `functools.reduce`~~
- ~~Decorators~~



Alt Text: a baby elephant with Week 7 vibes.
(Image Source)