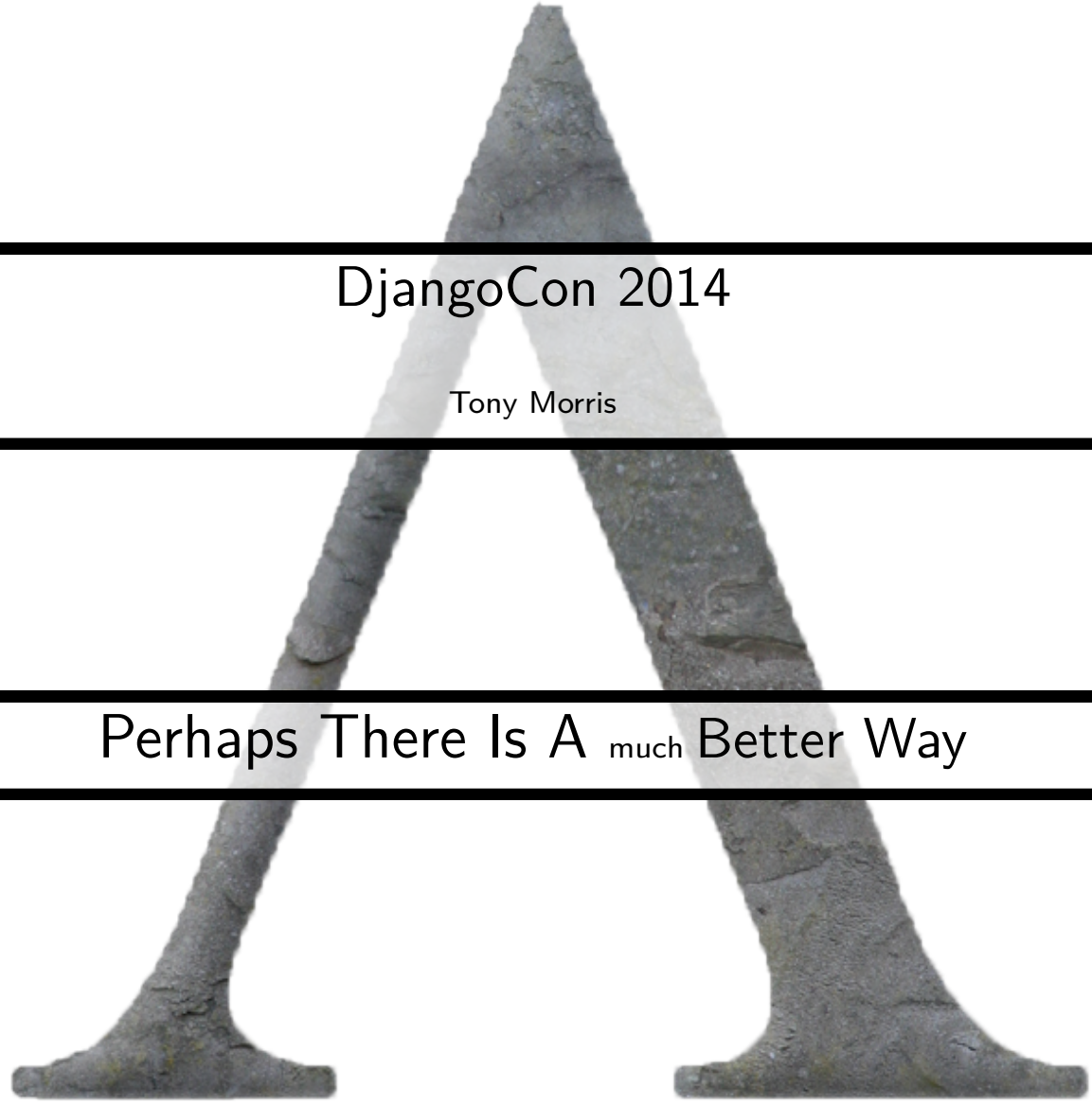


DjangoCon 2014

Tony Morris

Perhaps There Is A much Better Way

Goals

- Our existing common goals
- The goals for today

Goals
Our common goals

to implement software efficiently

- to arrive at an initial result as quickly as possible
- over time, to reliably arrive at results as quickly as possible (aka maintenance)
- a result is valid if the program accurately achieves our objective

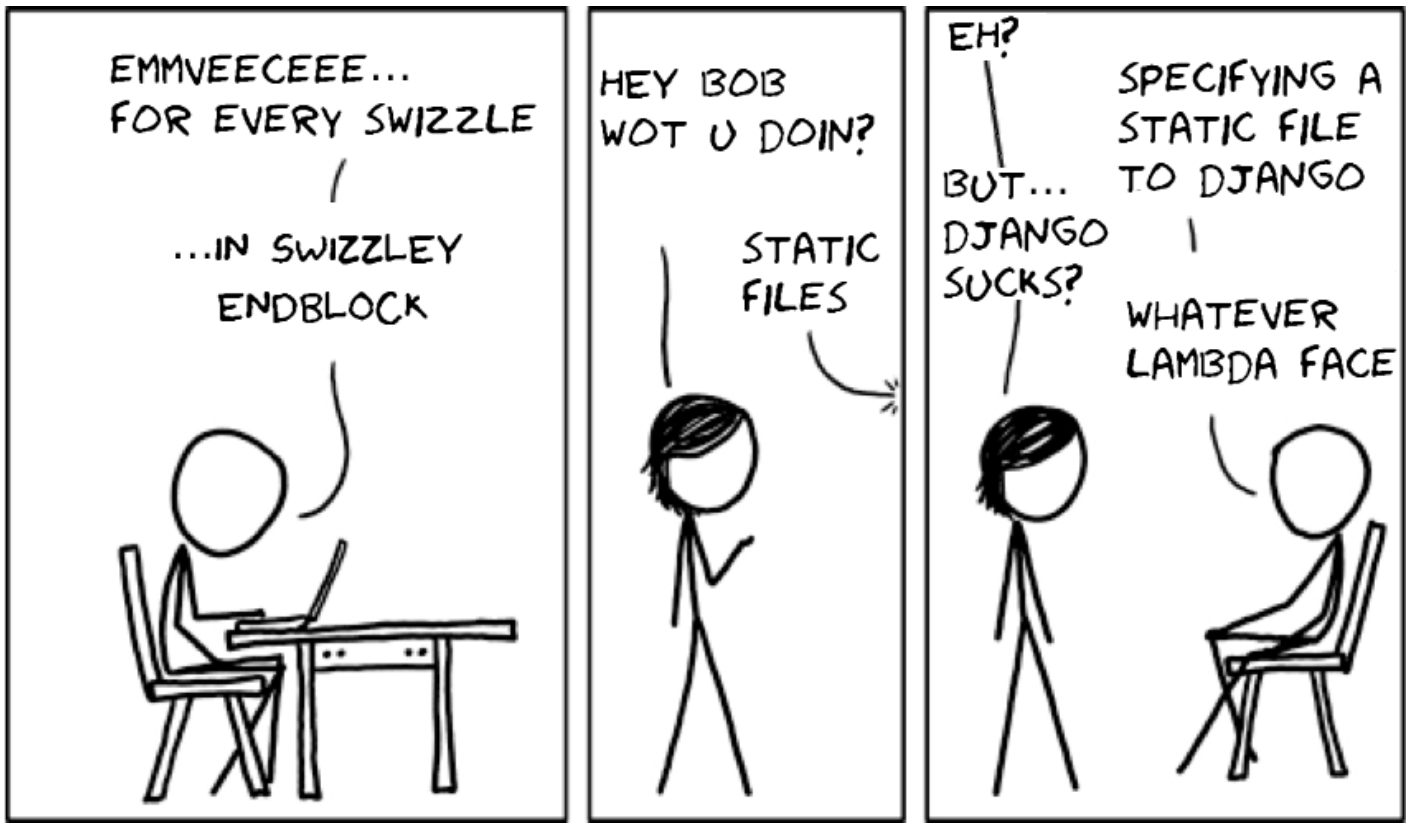
Goals
Our common goals

impertinent goals

- financial gain "but django makes me $$$!"
- motivate personal avidities

Goals
Our goals for today

"Why Django Sucks"



Goals
Our goals for today

I could talk all day about why django sucks

- and I'd be saying lots and lots of true things
- but not necessarily helpful things

NICTA

Goals
Our goals for today

I could talk all day about why django sucks

- and I'd be saying lots and lots of true things
- but not necessarily helpful things

NICTA

Goals
Our goals for today

The goal today

To equip you with new tools and perspective, with which to explore the question for yourself.

Along the way

We will visit some of the lies you have been told

Goals
Lies

Tacitly insidious ones

"Having to come to grips with Monads just isn't worth it for most people"

Confusingly insidious ones

"Imperative vs functional programming vs object-oriented programming"

Awkward ones

"The real world is mutable"

Goals
Lies

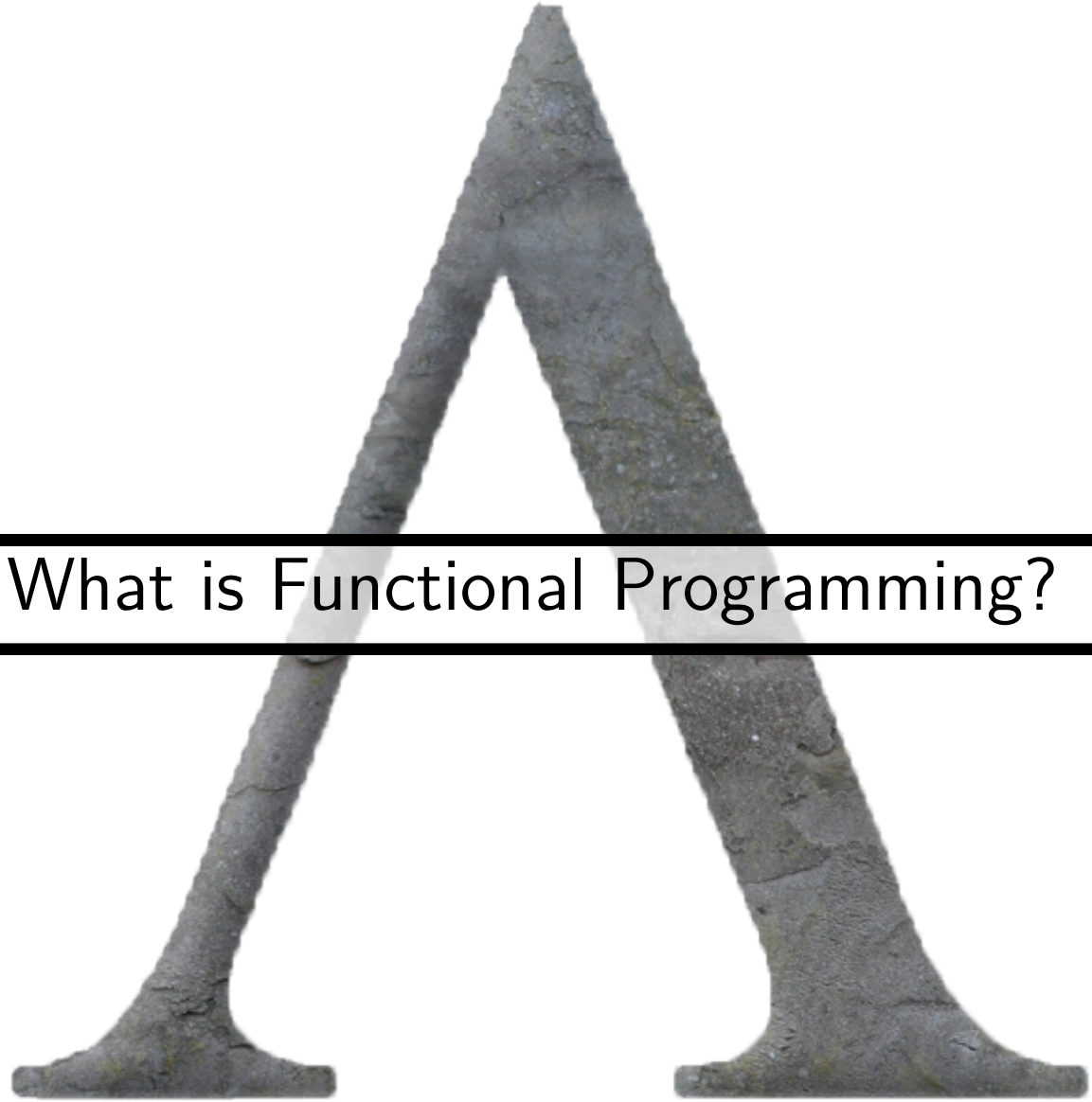
Funny ones

"Django: The Web framework for perfectionists with deadlines"

Goals

Summary

- What is functional programming?
- What does monad mean?
- Functional imperative programming
- Parametricity —types are documentation



What is Functional Programming?

Functional Programming

Split the question in two

- what does functional programming mean in principle?
- what are the consequences of this principle?

Functional Programming

What does functional programming mean?

- programming with functions
- yeah right, but what is a function?

Functional Programming

What does functional programming mean?

- programming with functions
- yeah right, but what is a function?

Functional Programming

A function

relates every argument to a result **and does nothing else**

Functions give rise to *referential transparency*

An expression `expr` is referentially transparent if in all programs `p`, all occurrences of `expr` in `p` can be replaced by the result assigned to `expr` without causing an observable effect on `p`.

Functional Programming

Functions give rise to *referential transparency*

```
def p():
  x = expression
  proc(x, x)
```

is `expression` referentially transparent?

Functions give rise to *referential transparency*

```
def p():
  # x = expression
  proc(expression, expression)
```

has this refactoring affected the program?

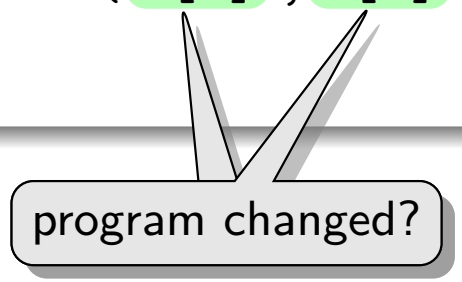
Referential Transparency

```
def print2(s, t):
  print(s)
  print(t)

def strpopthen():
  s = 'abcdef'
  x = s[0]
  print2(x,x)
```

referentially transparent?

Referential Transparency

```
def print2(s, t):
  print(s)
  print(t)

def strpopthen():
  s = 'abcdef'
  # x = s[0]
  print2(s[0],s[0])
```



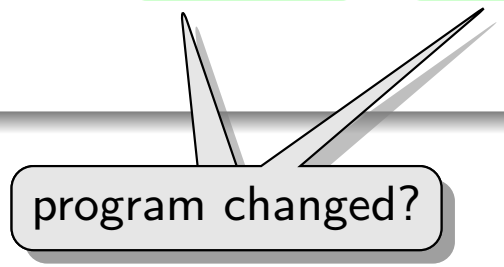
Referential Transparency

```
def print2(s, t):
  print(s)
  print(t)

def listpopthen():
  s = ['a','b','c','d','e','f']
  x = s.pop()
  print2(x,x)
```

referentially transparent?

Referential Transparency

```
def print2(s, t):
  print(s)
  print(t)

def listpopthen():
  s = ['a','b','c','d','e','f']
  # x = s.pop()
  print2(s.pop(),s.pop())
```



Functional Programming
What does functional programming mean?

The essence of functional programming

is the demand that expressions, **in general**, maintain referential transparency

Functional Programming
but why?

referential transparency gives rise to, **and monopolises**

- *equational reasoning*
  an essential tool for code readability
- *modularity*
  delineating concepts, building new programs from slightly smaller programs

Functional Programming
therefore

Functional Programming is

- a thesis, independent of any programming language
- to some extent, a programming language may provide the programmer assistance in achieving adherence to the thesis
- not anything more than this, despite what you may have been told

Functional Programming
therefore

Functional Programming is

- a thesis, independent of any programming language
- to some extent, a programming language may provide the programmer assistance in achieving adherence to the thesis
- not anything more than this, despite what you may have been told

Functional Programming
therefore

Functional Programming is

- a thesis, independent of any programming language
- to some extent, a programming language may provide the programmer assistance in achieving adherence to the thesis
- not anything more than this, despite what you may have been told

Well that raises an interesting question

Does python assist in achieving this objective?

No.

Functional Programming

This question has been explored and answered

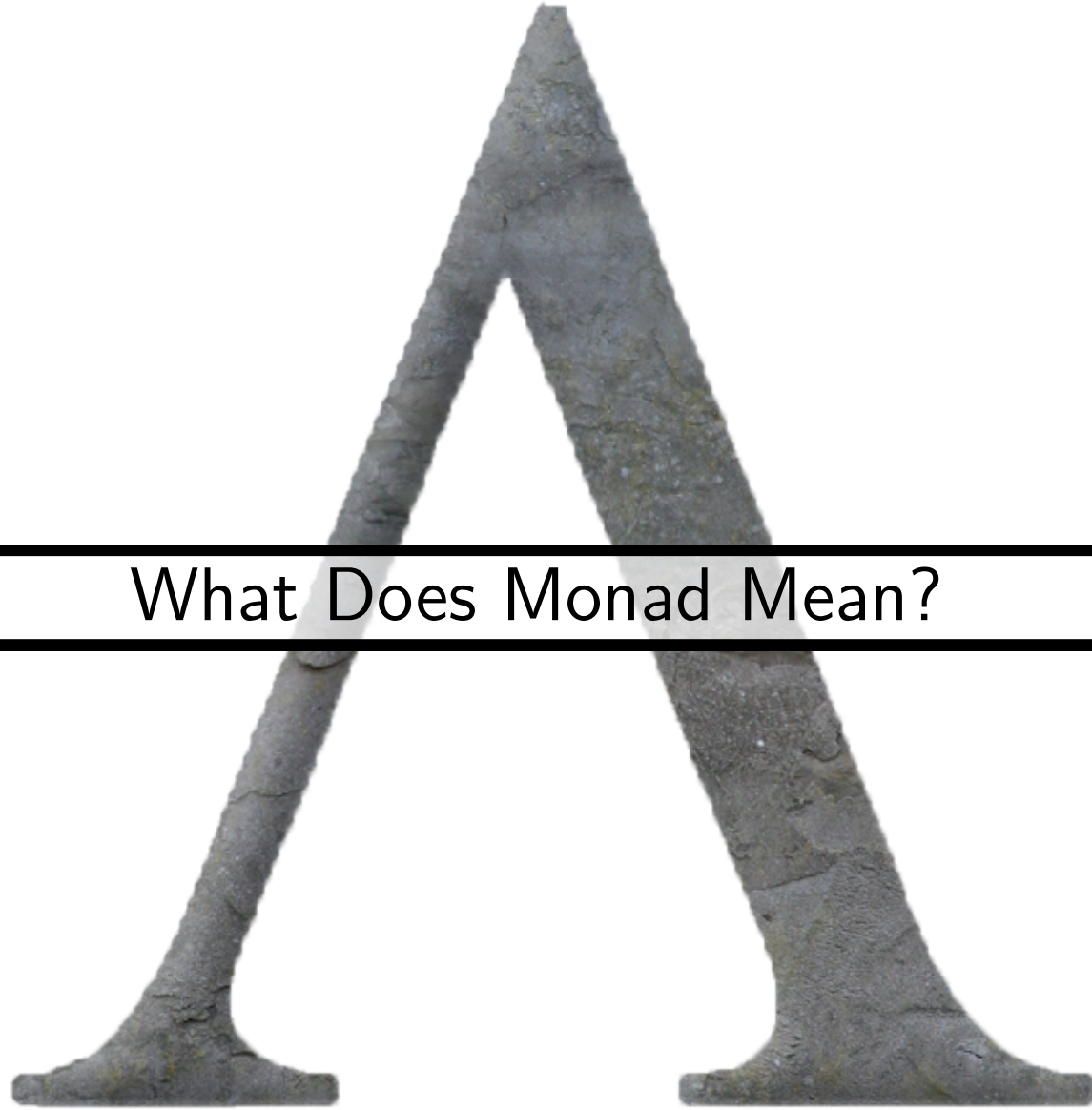
If anyone tells you that python supports functional programming to any (non-trivial) extent, **they are outright lying to you**

and I have the code to prove it

Functional Programming

As a consequence

Python demands that you, the programmer, forgo many of the most essential tools of progressive software development



What Does Monad Mean?

What Does Monad Mean?

Well if you google it

burritos, spacesuits or some shit like that

What Does Monad Mean?

I want to do three things

1. establish the principles of abstraction and what is required to exploit them
2. come to understand the relationship between monads and functional programming *(spoiler: there isn't one)*
3. arm you with the skills to recognise common myths

Principles and Goals of Abstraction

Construct a constraint

- minimise the requirements to satisfy the constraint to increase instances
- maximise potential for deriving operations as a consequence of satisfying the constraint

Principles and Goals of Abstraction

Trade-off between the two

- stronger constraint
  - fewer instances
  - more derived operations
- weaker constraint
  - more instances
  - fewer derived operations

Principles and Goals of Abstraction

The ultimate goal

Avoid repetition of the same work

Principles and Goals of Abstraction

Consequently

A proposed abstraction that loses in both directions is a *false economy* and must be efficiently discarded

Monad

Monad is another abstraction

- not expressible in degenerate static type systems
- difficult to demonstrate without a static type system

Monad

So sorry, but I must use a practical programming language now

```
class Monad f where
  (=<<) :: (a -> f b) -> f a -> f b
  unit  :: x -> f x
```

Monad

This abstraction has many instances (values for `f`)

- list
- continuations
- nullable values
- exception chaining
- state
- I/O actions
- argument threading
- logging
- *hundreds more*

Monad

This abstraction gives rise to many useful operations

- sequencing a list of effect values
  `[f a] -> f [a]`
- replicating an effect a given number of times
  `Int -> f a -> f [a]`
- *bazillions more*

Monad

We just saw

- the monad abstraction expressed as a constraint
- instances that satisfy the constraint
- operations that are derived from the constraint
- **Do not conflate these**

Monad

We just saw

- the monad abstraction expressed as a constraint
- instances that satisfy the constraint
- operations that are derived from the constraint
- **Do not conflate these**

Monad

We just saw

- the monad abstraction expressed as a constraint
- instances that satisfy the constraint
- operations that are derived from the constraint
- **Do not conflate these**

Monad

We just saw

- the monad abstraction expressed as a constraint
- instances that satisfy the constraint
- operations that are derived from the constraint
- **Do not conflate these**

Abstraction nomenclature

Other abstractions trade off along the two competing principles

- covariant functor
- applicative functor
- semigroupoid
- comonad
- profunctor
- monoid
- *hundreds more*

Now that we know this

We can do some mythbusting

- Monads are for side-effects
- Monads are for functional programming only
- Monads are for doing I/O
- Monads don't apply to my programming tasks
- I use Python, so monads won't help me as much

Now that we know this

We can do some mythbusting

- Monads are for side-effects
- Monads are for functional programming only
- Monads are for doing I/O
- Monads don't apply to my programming tasks
- I use Python, so monads won't help me as much

Now that we know this

We can do some mythbusting

- Monads are for side-effects
- Monads are for functional programming only
- Monads are for doing I/O
- Monads don't apply to my programming tasks
- I use Python, so monads won't help me as much

Now that we know this

We can do some mythbusting

- Monads are for side-effects
- Monads are for functional programming only
- Monads are for doing I/O
- Monads don't apply to my programming tasks
- I use Python, so monads won't help me as much

Now that we know this

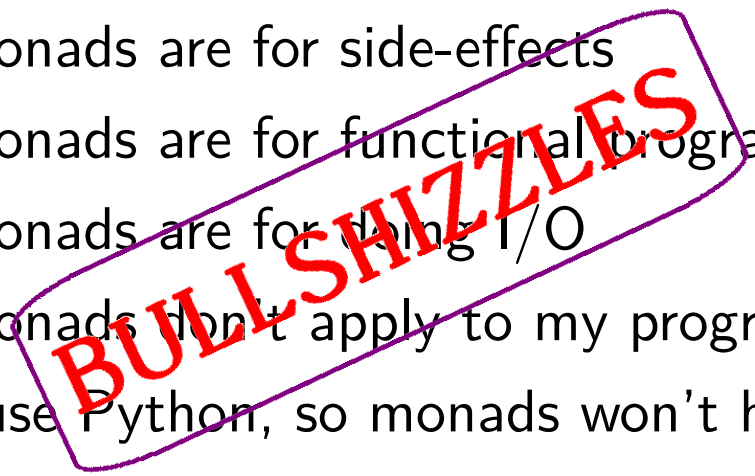
We can do some mythbusting

- Monads are for side-effects
- Monads are for functional programming only
- Monads are for doing I/O
- Monads don't apply to my programming tasks
- I use Python, so monads won't help me as much

Now that we know this

We can do some mythbusting

- Monads are for side-effects
- Monads are for functional programming only
- Monads are for doing I/O
- Monads don't apply to my programming tasks
- I use Python, so monads won't help me as much



Bullshizzles

Perfidious Seduction

You will be invited to believe these things. Decide wisely.

Abstraction and Python

Python

Python achieves abstraction using dynamic structural-typing[a]

[a] aka duck typing

Abstraction and Python

Further to this

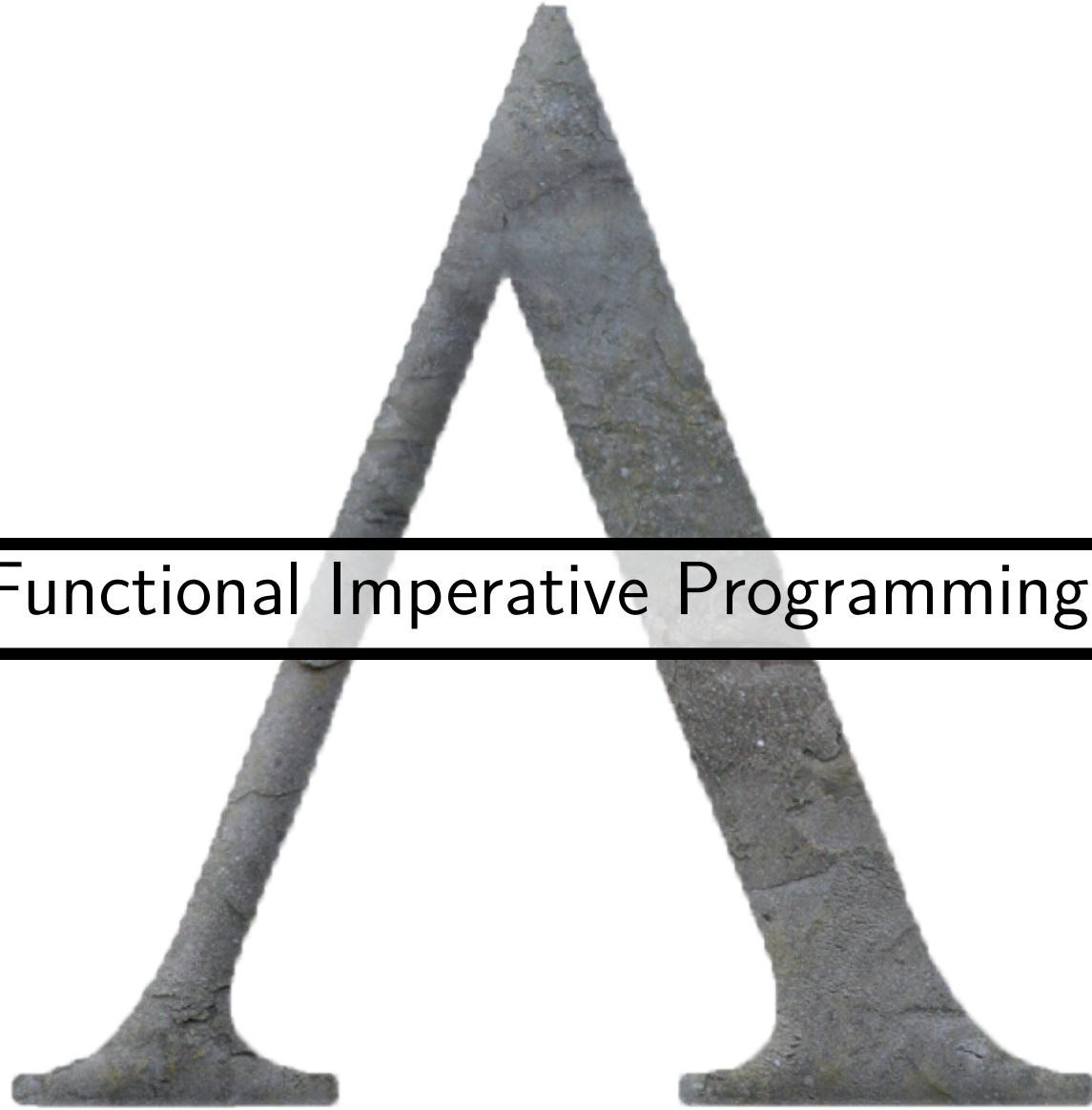
You will rarely see even the most fundamental abstractions in these systems

Abstraction and Python

Certainly

You will never see any non-trivial abstraction *(above those already mentioned)*

Why might this be?



Functional Imperative Programming

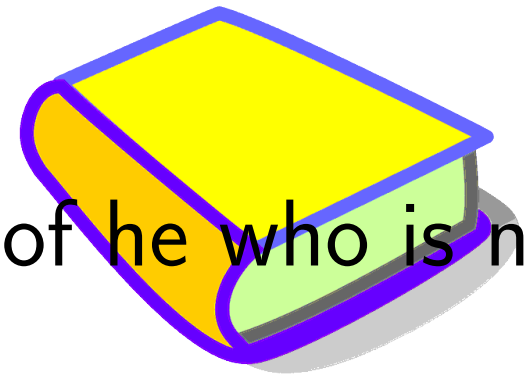
Functional v Imperative v OOP

The fallacy of false compromise

A note on OOP

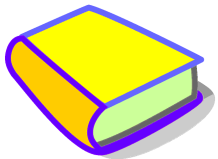
I am going to dismiss Object-Oriented Programming, because I don't know what it is and neither do you

Functional v Imperative

The parable of  he who is not even wrong

Functional v Imperative

The parable of he who is not even wrong

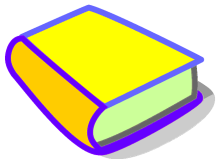


Someone in a pub once said to me, not too long ago

> *Hi my name is Wiggleydoo and I am the Chief Python Wippedy-wop for Hoopdiddy-zip*

Functional v Imperative

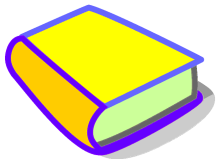
The parable of he who is not even wrong



and I thought to myself, “oh yeah that’s nice”

Functional v Imperative

The parable of he who is not even wrong

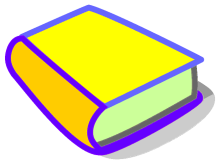


and then this happened

> *Functional programming is great and all, but I only use state where it is appropriate. You know. . . when the problem demands stateful things.*

Functional v Imperative

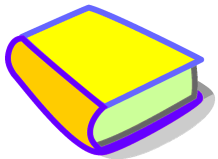
The parable of he who is not even wrong



and it all came back to me

Functional v Imperative

The parable of he who is not even wrong

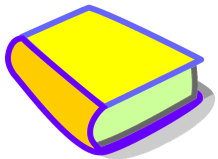


Fact

There is no such thing as an "inherently stateful" computation or algorithm

Functional v Imperative

The parable of he who is not even wrong



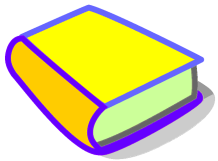
So I called shenanigans on that

but what about those algorithms that demand imperative programming?

NICTA

Functional v Imperative

The parable of he who is not even wrong

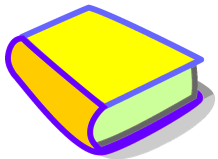


Church-Turing Thesis

Many people do imperative programming using pure-functional programming all day, every day

This can be achieved for every program that can possibly exist

Functional v Imperative
The parable of he who is not even wrong

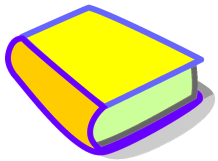


"but how?"

At this point I am stumped. "Casually . . . er neatly?"

Functional v Imperative

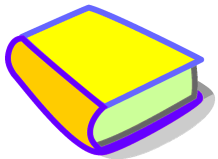
The parable of he who is not even wrong



Recognising my confoundment

My friend pulled out some imperative pure-functional production code that had been written at a bank

Functional v Imperative
The parable of he who is not even wrong



Escaping the state of delirium

The discussion then quickly turned to beer

Functional v Imperative

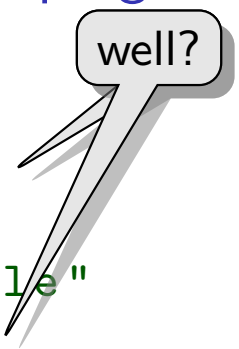
Here is an imperative Haskell program

```
program = do
  a <- readFile "file"
  print a
  writeFile "cods!" "file"
  b <- readFile "file"
  print b
```

Functional v Imperative

Here is an imperative Haskell program

```
program = do
  a <- readFile "file"
  print a
  writeFile "cods!" "file"
  b <- readFile "file"
  print b
```

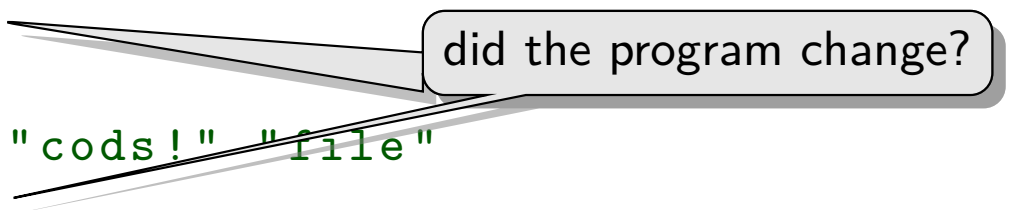


NICTA

Functional v Imperative

Here is an imperative Haskell program

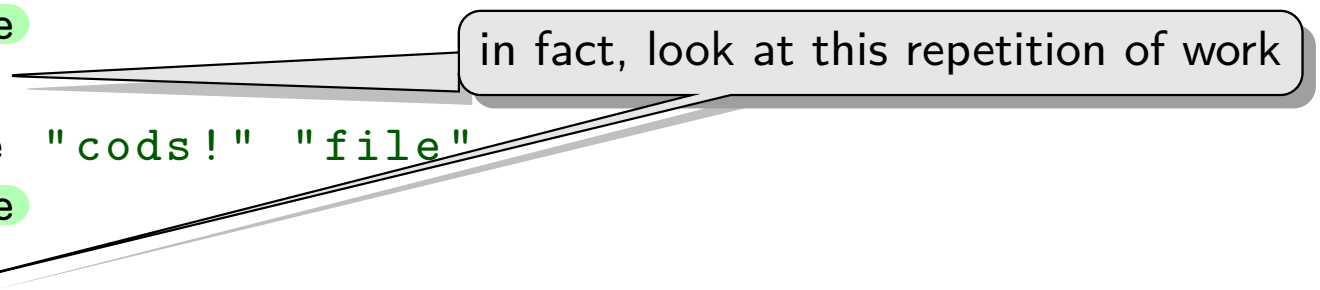
```
file =
  readFile "file"

program = do
  a <- file
  print a
  writeFile "cods!" "file"
  b <- file
  print b
```



Functional v Imperative

Here is an imperative Haskell program

```
file =
  readFile "file"

program = do
  a <- file
  print a
  writeFile "cods!" "file"
  b <- file
  print b
```



Functional v Imperative

Here is an imperative Haskell program

```
printfile = do
  f <- readFile "file"
  print f

program = do
  printfile
  writeFile "cods!" "file"
  printfile
```

Functional v Imperative

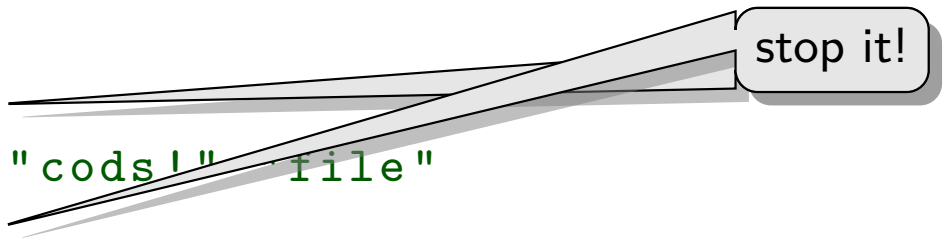
Did I mention that functional programming is

Don't Repeat Yourself without the duplicity?

Functional v Imperative

Here is an imperative Haskell program

```
printfile = do
  f <- readFile "file"
  print f

program = do
  printfile
  writeFile "cods!" "file"
  printfile
```



Functional v Imperative

Here is an imperative Haskell program

```
a >.> b = do
  a
  b
  a

printfile = do
  f <- readFile "file"
  print f

program =
  printfile >.> writeFile "cods!" "file"
```

Functional v Imperative

We can do this equational reasoning on imperative programs

because we are functional programming

Functional v Imperative

- Functional Programming
- or Imperative Programming
- **but never both**

Functional v Imperative

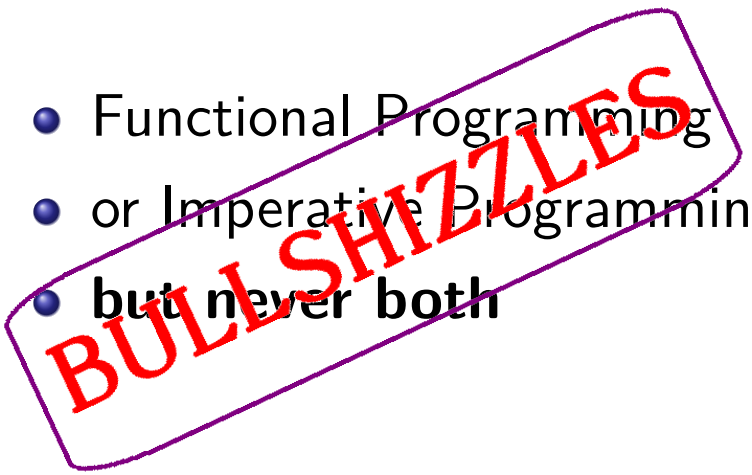
- Functional Programming
- or Imperative Programming
- but never both

Functional v Imperative

- Functional Programming
- or Imperative Programming
- **but never both**

Functional v Imperative

- Functional Programming
- or Imperative Programming
- **but never both**



NICTA

Functional v Imperative

Functional programming is

just a not ridiculous means of imperative programming

Functional v Imperative

To further help demonstrate this point

- pure-functional random value library using the C# programming language
- pure-functional RDBMS library using the Java programming language
- pure-functional terminal I/O programs using the Ruby programming language

Functional v Imperative

To further help demonstrate this point

- pure-functional random value library using the C# programming language
- pure-functional RDBMS library using the Java programming language
- pure-functional terminal I/O programs using the Ruby programming language

Functional v Imperative

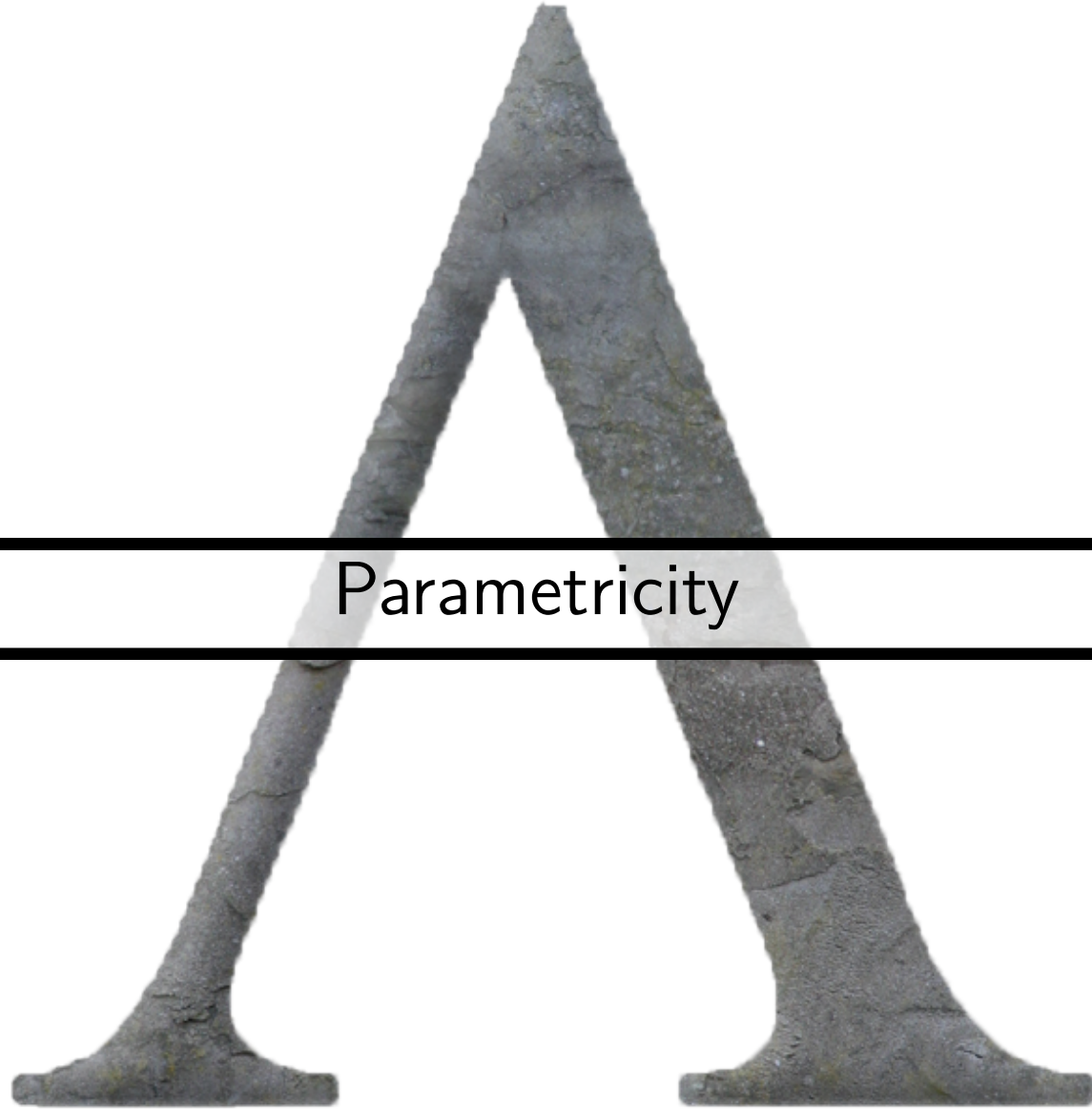
To further help demonstrate this point

- pure-functional random value library using the C# programming language
- pure-functional RDBMS library using the Java programming language
- pure-functional terminal I/O programs using the Ruby programming language

Functional v Imperative

"I don't think the benefits of enforcing side-effect-free code (even optionally) make up for the many work-arounds you have to use to get anything done in the real world."

"[Functional Programming makes] the code pretty unreadable for people who aren't used to functional"



Parametricity

Parametricity

Parametricity, or Theorems for Free, is

a technique described by Wadler[Wad89] that gives rise to many practical consequences.

Parametricity

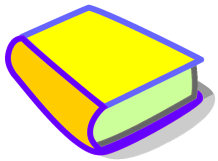
One practical consequence of parametricity is that

Types, by exploiting generalisation, may be utilised to document code and that documentation never goes out of date.

Parametricity

Many nutty things have been said about static typing . . .

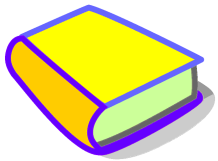
Parametricity



. . . including this one time

a neurotic colleague insisted that static typing is terrible . . .

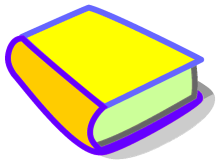
Parametricity



. . . because you have to write messy types all the time!

```
int x
x = 3
String y
y = "abc"
(String, Int) a
a = ("xyz", 9)
[Int] b
b = [1,4,9,16,x]
```

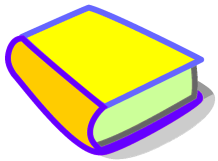
Parametricity



So I took the Python source file

```
x = 3
y = "abc"
a = ("xyz", 9)
b = [1,4,9,16,x]
```

Parametricity



. . . and loaded it into the Glasgow Haskell Compiler repl

```
> ln -s soneat.py notypes.hs
> ghci notypes.hs
```

Parametricity

The goal is to tell you about

a more subtle, and rarely discussed, advantage that is available by appropriately exploiting types

not so much to engage the static typing debate itself

Philip Wadler [Wad89] tells us:

> *Write down the definition of a polymorphic function on a piece of paper. Tell me its type, but be careful not to let me see the function's definition. I will tell you a theorem that the function satisfies.*
>
> *The purpose of this paper is to explain the trick.*

Totality

Fast and loose reasoning is morally correct

Danielsson, Hughes, Jansson & Gibbons [DHJG06] tell us:

> *Functional programmers often reason about programs as if they were written in a total language, expecting the results to carry over to non-total (partial) languages. We justify such reasoning.*

Parametricity

What if I told you

There exists a function that takes a list of some element type and returns a list of elements of that same type?

```
function :: [a] -> [a]
```

Parametricity

I observe that

there is nothing else to know about this element type

Its declared structure is *universally quantified*

Parametricity

On this information alone, I can $\therefore$ conclude that

Theorem: every element in the result appears in the input

Morally Correct?

What does it mean to "reason as if we were in a total language, expecting the results to carry over to non-total languages"?

Parametricity
Fast and Loose Reasoning

It means we can "morally" exclude logical-inconsistencies $\perp$

- Type-case (e.g. `is`)
- Type-cast
- Type-value (e.g. `type`)
- unsafe universal functions e.g. `str`, `int`
- side-effects
- `None` or `null`
- exceptions
- Infinite recursion

Parametricity
Fast and Loose Reasoning

Some non-total systems do not have these $\perp$ values anyway

- ~~Type-case e.g. `(is)`~~
- ~~Type-cast~~
- ~~Type-value e.g. `(type)`~~
- ~~unsafe universal functions e.g. `str`, `int`~~
- ~~side-effects~~
- ~~`None` or `null`~~
- exceptions
- Infinite recursion

On the basis of fast and loose reasoning

Can we invalidate the theorem, **every element in the result appears in the input**?

```
function :: [a] -> [a]
```

Parametricity

OK, so we have machine-checked documentation, but

How do we know what this function does exactly?

Parametricity

We can write unit tests

but not your laborious and clumsy kind

Parametricity

Specification-based tests or universally quantified properties[CH11]

- $\forall$ `x. function [x] == [x]`
- $\forall$ `x y. function (x ++ y) == function y ++ function x`
- Reminder: `function :: [a] -> [a]`
- Now what does our `function` do?

NICTA

Parametricity

Specification-based tests or universally quantified properties[CH11]

- $\forall$ `x. function [x] == [x]`
- $\forall$ `x y. function (x ++ y) == function y ++ function x`
- Reminder: `function :: [a] -> [a]`
- Now what does our `function` do?

Parametricity

More theorems for free

- comp :: (a -> b) -> (b -> c) -> (a -> c)
- choose :: (a, a) -> a

NICTA

Parametricity

More theorems for free

- comp :: (a -> b) -> (b -> c) -> (a -> c)
- choose :: (a, a) -> a

A note on your bravery

Introspection

So

I typed “functional programming for python” into an internet search . . .

Bravery

Functional Programming . . .

The designers [of functional programming languages] . . . choose to emphasize one particular approach to programming  This is . . . difficult to write programs that use a different approach  Other languages are multi-paradigm languages that support several different approaches. Lisp, C++, and Python are multi-paradigm . . . can write programs . . . procedural, object-oriented, or functional in all of these languages 

Bravery

Some languages . . .

Some [functional programming] languages don't even have assignment statements such as a=3 or c=a+b, but it's difficult to avoid all side effects Printing to the screen or writing to a disk file are side effects, for example

NICTA

Bravery

Formal provability

A theoretical benefit is that it's easier to construct a mathematical proof that a functional program is correct . . . Unfortunately, proving programs correct is largely impractical and not relevant to Python software

Bravery

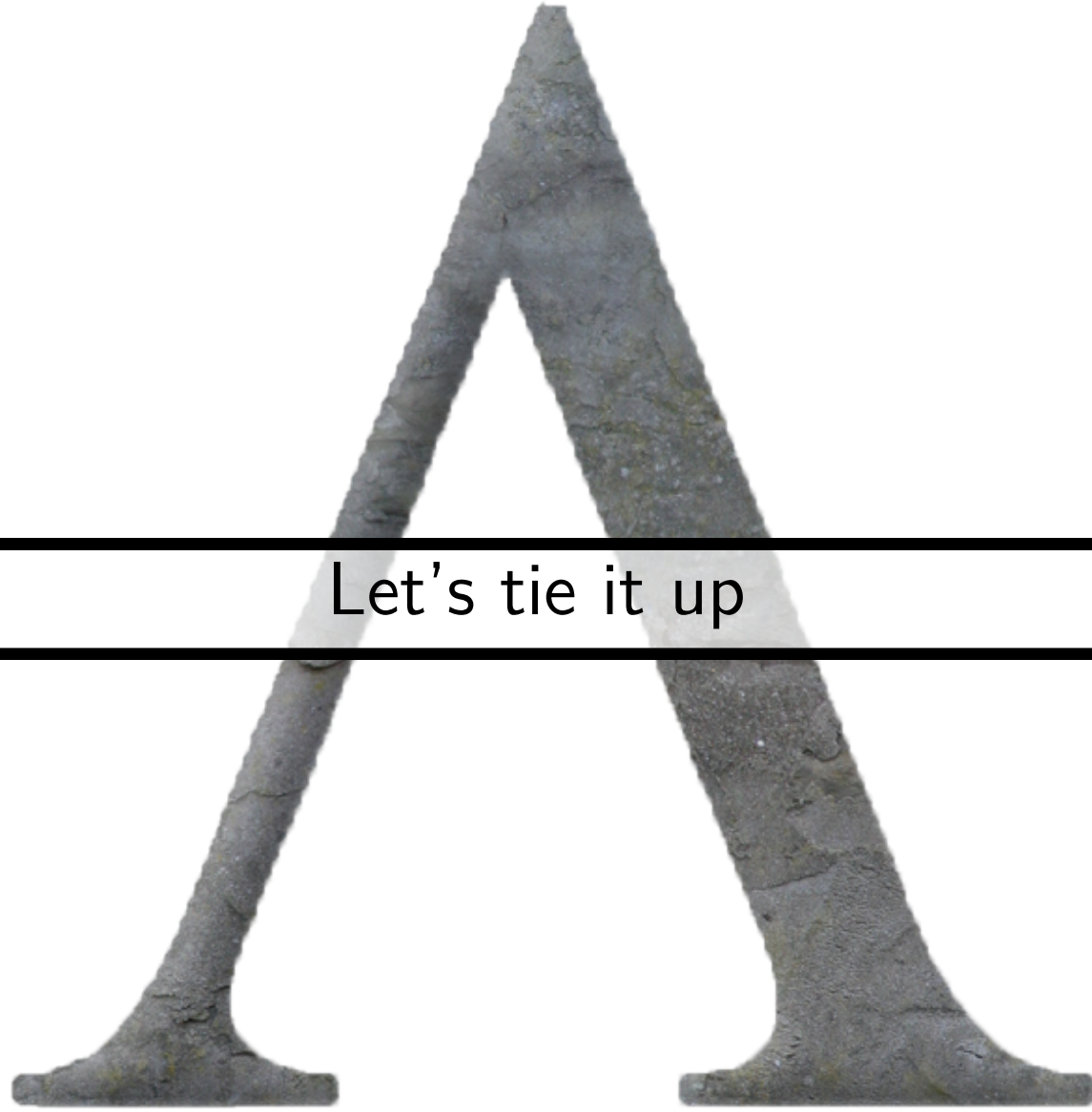
iterators

a Python language feature that's an important foundation for writing functional-style programs: **iterators** 

Introspection

Just when

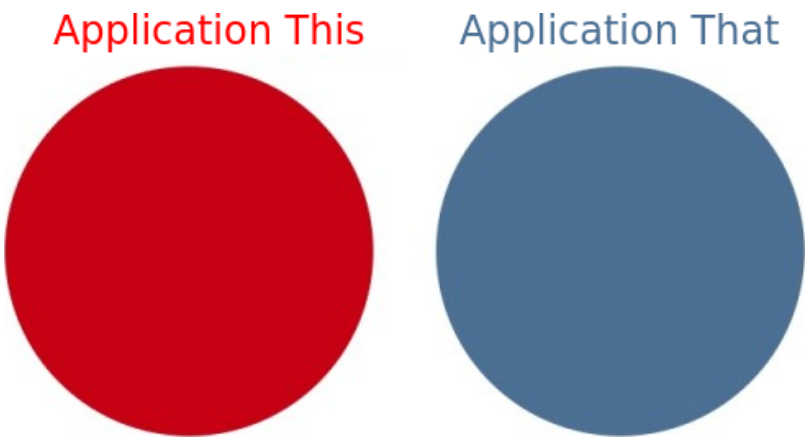
Just when I thought I'd seen enough, I realised **just how brave you all are**



Let's tie it up

Django

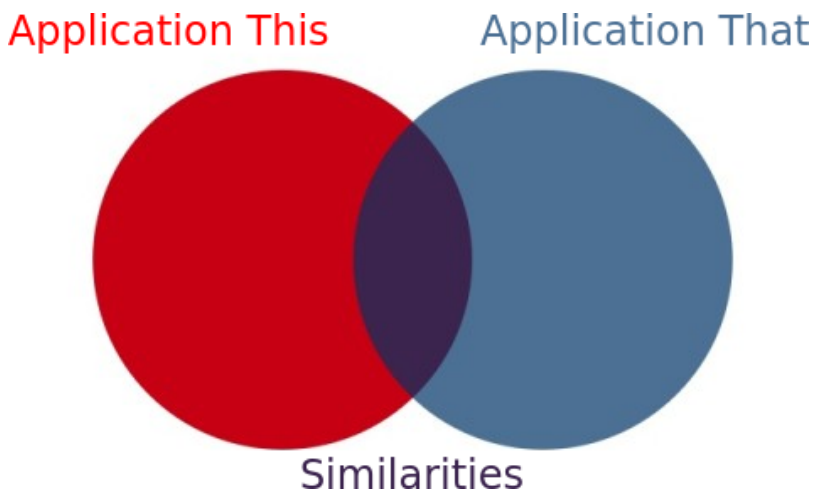
The primary purpose of frameworks like Django is to observe similarities in otherwise disjoint applications . . .



NICTA

Django

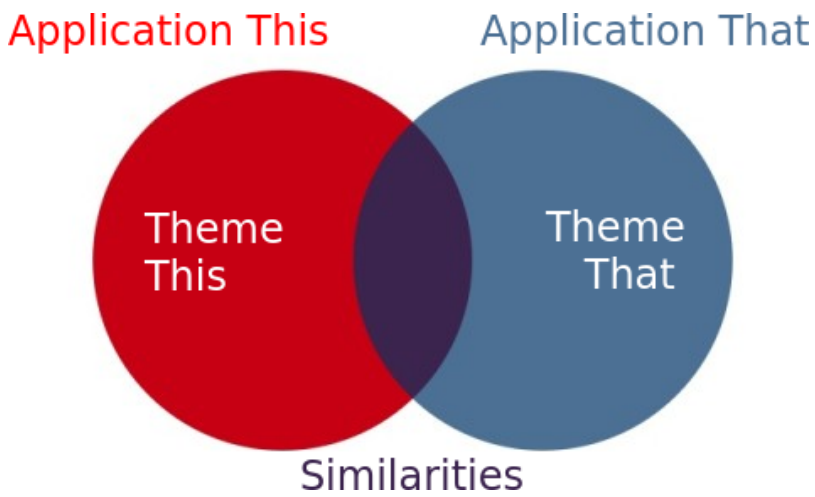
. . . and provide library support for those similarities, while distingushing from differences.



NICTA

Django

For example, Django *themes* or *templates*



Bugs

Maintenance issues

Maintenance issues come about when software components leak over these boundaries

Equational Reasoning

We have a name for this delineation of concepts

Equational reasoning

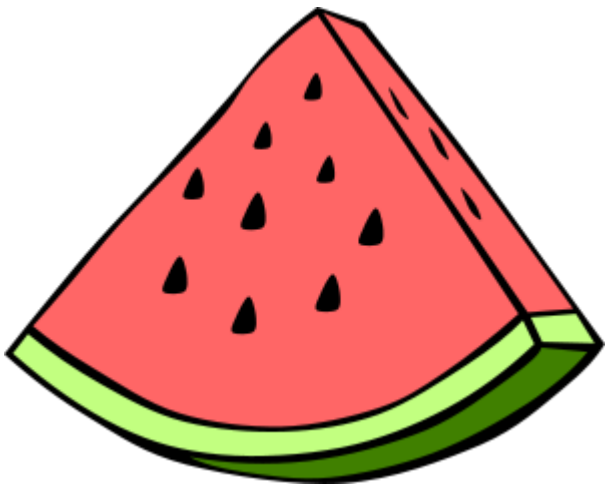
This is the essence of functional programming!

Tool Support

OK, so what support do our tools provide us for exploiting this?

Tool Support

0



"Is it really such a big deal to rewrite your function to use a loop?"

NICTA

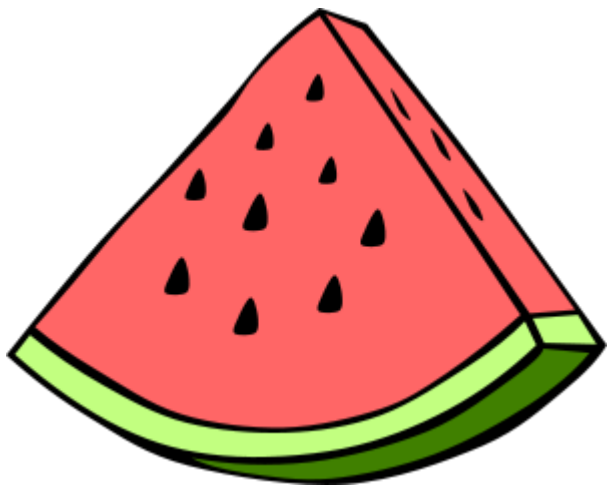
Tool Support

Oh

Then what about parametricity? Abstraction?

NICTA

Again, 0



Tool Support

More to the point

There is a **huge** amount of code that I cannot write without these tools . . .

Tool Support

. . . and I suspect

that nobody can, because I never see it. Have you?

NICTA

So here's why

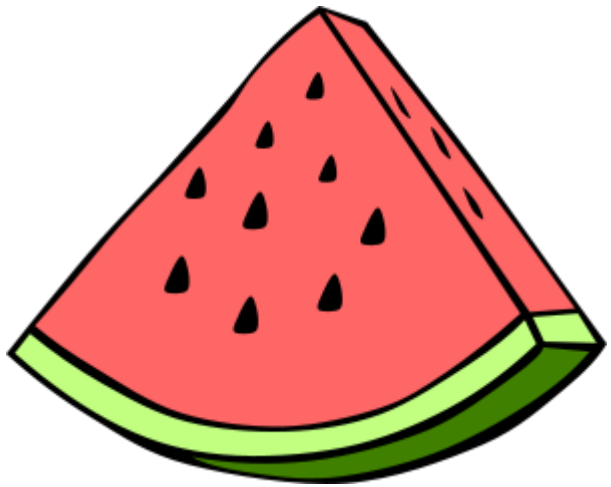
If you accept

- Equational reasoning
- Parametricity
- Abstraction

are necessary, rudimentary tools to programming,

So here's why

Then you also accept



NICTA

How?

How might I learn to exploit these tools?

Diversify

- Total programming (Agda, Coq, Idris)
- Pure functional programming (Haskell)

TL;DR

TL;DR

[–] **ruinercollector** 9 points 10 months ago

I have found that "Pythonic" means:

"I'm ignoring years of academic/industry knowledge about programming and languages because I don't understand them."

permalink **save** **parent** **give gold**

References


Koen Claessen and John Hughes, *Quickcheck: a lightweight tool for random testing of haskell programs*, Acm sigplan notices **46** (2011), no. 4, 53–64.


Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons, *Fast and loose reasoning is morally correct*, ACM SIGPLAN Notices, vol. 41, ACM, 2006, pp. 206–217.


Philip Wadler, *Theorems for free!*, Proceedings of the fourth international conference on Functional programming languages and computer architecture, ACM, 1989, pp. 347–359.

Licence Attributions

- Some images used in this presentation are attributed to Joshua Morris
- Some images used in this presentation are attributed to XKCD released under the CC BY-NC licence
- Some images used in this presentation are attributed to NatSilva released under the CC BY-NC-ND licence
- Some ideas in BeamerTitleSlide by Daniel Falster are used in this presentation