SNAKES & LADDERS: A TASTE OF SCALA FOR PYTHONISTAS

SCALA (EPFL, LAUSANNE):

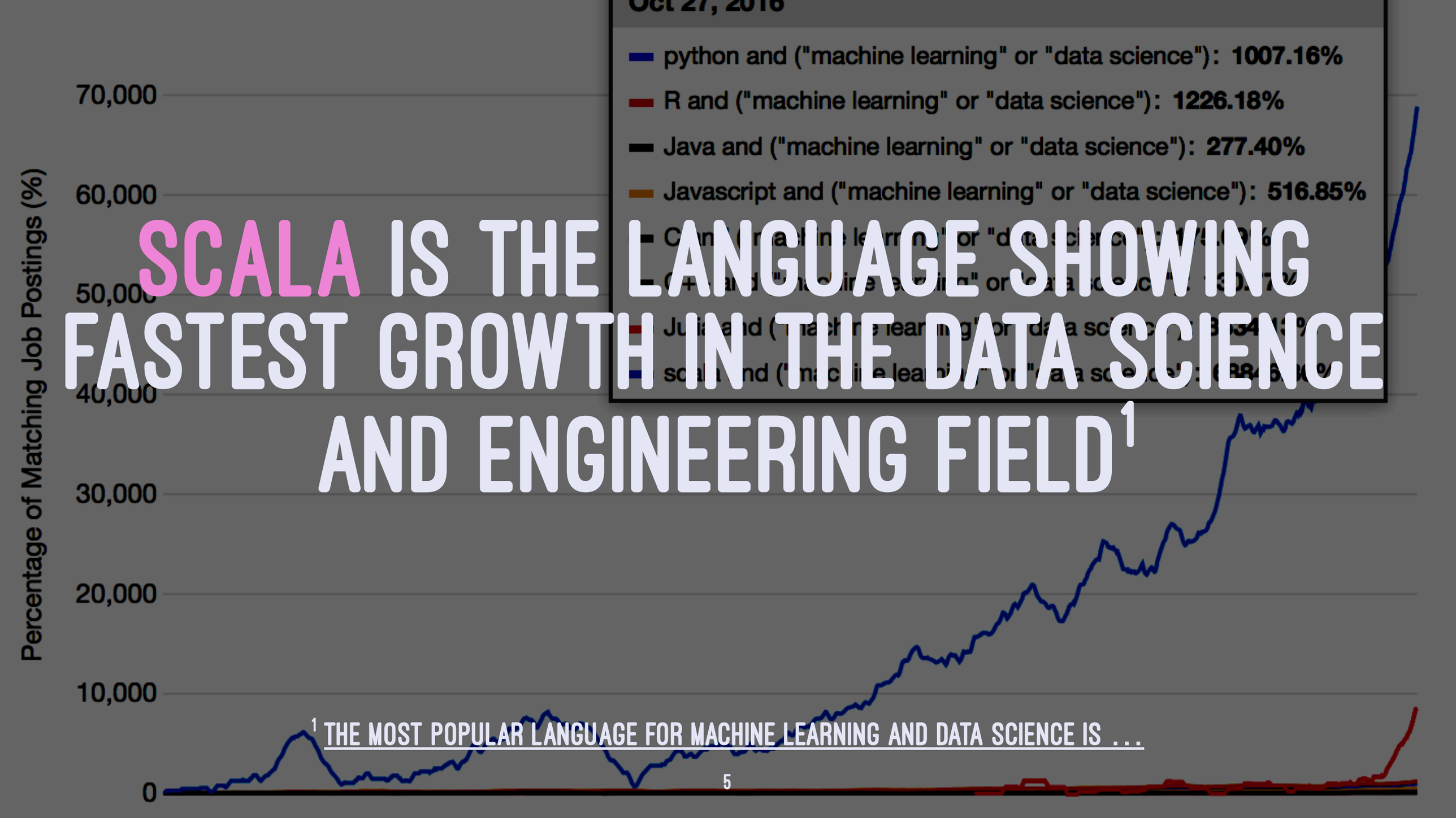
FUNCTIONAL. OO. ADVANCED TYPE SYSTEM

PYTHON(CWI AMSTERDAM):

MULTI-PARADIGM

CAN WE GO FROM ONE TO THE OTHER?

WHY **SHOULD** WE?



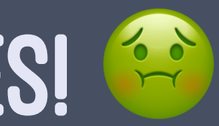
SCALA IS THE LANGUAGE SHOWING FASTEST GROWTH IN THE DATA SCIENCE AND ENGINEERING FIELD[1]

[1] THE MOST POPULAR LANGUAGE FOR MACHINE LEARNING AND DATA SCIENCE IS ...

SOME REASONS WHY SCALA IS TRENDY

1. RUNS IN THE JVM (INTEROPERATES WITH JAVA 🤢)
2. APACHE SPARK IS IN SCALA (CAN BE USED FROM PYTHON 😍)
3. SCALA DEVELOPERS EARN MORE 🤑

BUT...

1. IT RUNS ON THE JVM... YIKES! 
2. SCALA LOOKS WEIRD 
3. THERE IS A LOT OF JARGON IN SCALA 

SO...

TODAY WE'LL SEE HOW SCALA IS NOT THAT WEIRD IF YOU KNOW (WEIRD) PYTHON

WHOAMI

- RUBEN BERENGUEL
- PHD IN MATHEMATICS
- (BIG) DATA CONSULTANT
- WRITING SOME PYTHON FOR THE PAST 12 YEARS
- STARTED WITH SCALA A YEAR AGO
- RIGHT NOW AT AFFECTV IN LONDON (WE ARE HIRING!)


SCALA IS STATICALLY
TYPED: NO int + str IN
PRODUCTION

WE HAVE MYPY: PYTHON'S STATIC TYPE CHECKER!

```python
# mypy_example.py:
from typing import List

answer = 42  # type: int

dont = "panic"  # type: str

again = ["don't", "panic"]  # type: List[str]
```

TYPECHECK THE FILE WITH: `mypy mypy_example.py`: 👍

```python
# mypy_bad.py
from typing import List

answer = 42  # type: str
dont = "panic"  # type: int
again = ["don't", "panic"]  # type: List[float]
```

TYPECHECK THE FILE WITH: `mypy mypy_example.py`: 👎

```
mypy_bad.py:3: error: Incompatible types in assignment
    (expression has type "int", variable has type "str")
mypy_bad.py:5: error: Incompatible types in assignment
    (expression has type "str", variable has type "int")
mypy_bad.py:7: error: List item 0 has incompatible type "str"
mypy_bad.py:7: error: List item 1 has incompatible type "str"
```

SCALA LETS YOU DO FANCY TYPE-LEVEL STUFF. BUT THE BASICS ARE THE SAME AS IN MYPY

```
scala> val answer: Int = 42
answer: Int = 42

scala> val dont: String = "panic"
dont: String = panic

scala> val again: List[String] = List("don't", "panic")
again: List[String] = List(don't, panic)

scala> val noAnswer: Int = "panic"
<console>:11: error: type mismatch;
 found   : String("panic")
 required: Int
       val noAnswers: Int = "panic"
```

CAN ALSO INFER TYPES

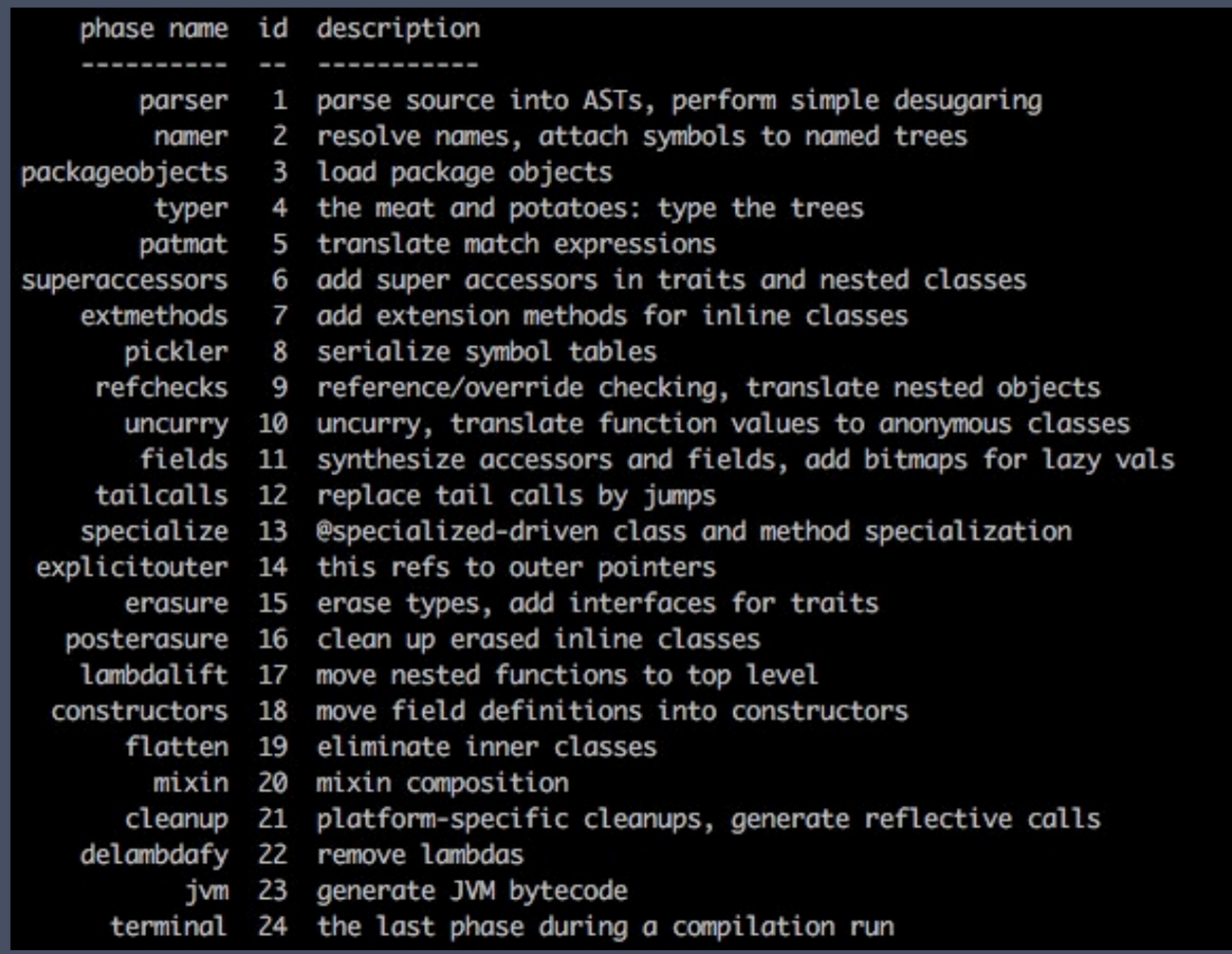
```
scala> val answer = 42
answer: Int = 42

scala> val dont = "panic"
dont: String = panic

scala> val again = List("don't", "panic")
again: List[String] = List(don't, panic)
```

TYPE INFERENCE IS DONE BY THE TYPER

`typer` 4 THE MEAT AND POTATOES: TYPE THE TREES

THE SCALA COMPILER HAS 24 PHASES

(SLOOOW)



phase name	id	description
parser	1	parse source into ASTs, perform simple desugaring
namer	2	resolve names, attach symbols to named trees
packageobjects	3	load package objects
typer	4	the meat and potatoes: type the trees
patmat	5	translate match expressions
superaccessors	6	add super accessors in traits and nested classes
extmethods	7	add extension methods for inline classes
pickler	8	serialize symbol tables
refchecks	9	reference/override checking, translate nested objects
uncurry	10	uncurry, translate function values to anonymous classes
fields	11	synthesize accessors and fields, add bitmaps for lazy vals
tailcalls	12	replace tail calls by jumps
specialize	13	@specialized-driven class and method specialization
explicitouter	14	this refs to outer pointers
erasure	15	erase types, add interfaces for traits
posterasure	16	clean up erased inline classes
lambdalift	17	move nested functions to top level
constructors	18	move field definitions into constructors
flatten	19	eliminate inner classes
mixin	20	mixin composition
cleanup	21	platform-specific cleanups, generate reflective calls
delambdafy	22	remove lambdas
jvm	23	generate JVM bytecode
terminal	24	the last phase during a compilation run

CASE CLASSES ARE COOL

```
scala> case class Person(name: String, age: Int) {
     | def greet = s"Hello, $name"
     | }
defined class Person

scala> val someone = Person("Joe Doe", 42)
someone: Person = Person(Joe Doe,42)
```

Q: IS GREET A FUNCTION?

A: IN **SCALA**, **EVERYTHING** IS **(KIND OF)** A FUNCTION[2]

[2] EVERYTHING IS AN EXPRESSION (HENCE HAS A VALUE). AND THAT VALUE IS ACTUALLY AN OBJECT 🤯

BECAUSE OF PATTERN MATCHING

```
// someone: Person = Person(Joe Doe,42)
scala> someone match {
     | case Person("Joe Doe", _) => someone.greet
     | case _ => "Who are you?"
     | }
//res0: String = Hello, Joe Doe

// someoneElse: Person = Person(Jane Doe,43)
scala> someoneElse match {
     | case Person("Joe Doe", _) => someoneElse.greet
     | case _ => "Who are you?"
     | }
//res1: String = Who are you?
```

SADLY WE DON'T HAVE ANYTHING SIMILAR IN PYTHON

(BUT WE LAUGH LAST: WHEN SCALA DEVS NEED TO MOCK OBJECTS 😬)

IN PYTHON WE CAN USE ABSTRACT BASE CLASSES TO DEFINE HOW CLASSES NEED TO BEHAVE

(ENFORCED AT INSTANTIATION TIME)

```python
from abc import ABCMeta, abstractmethod

class Singer(metaclass=ABCMeta):
    @abstractmethod
    def sing(self, tune):
        pass

class GuitarPlayer(metaclass=ABCMeta):
    @abstractmethod
    def rock(self, tune):
        pass

class Rocker(Singer, GuitarPlayer):
    def sing(self, tune):
        return f"YEAH {tune}"

    def rock(self, tune):
        return f"🎸 {tune} 🎸"

class BadRocker(Singer):
    pass
```

```
>>> from abstract_example import Rocker
>>> gene_simmons = Rocker()

>>> gene_simmons.sing("Psycho Circus")
YEAH Psycho Circus

>>> gene_simmons.rock("Strutter")
🎸 Strutter 🎸
```

```
>>> from abstract import BadRocker
>>> foo = BadRocker()

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class BadRocker
    with abstract methods sing
```

WITH ABCS & MYPY WE CAN ENSURE TYPE SAFETY

```python
def max_rock(rocker: Rocker, tune: str) -> str:
    return f"{rocker.rock(rocker.sing(tune))}"

gene_simmons = Rocker()
max_rock(gene_simmons, "I was made for loving you")
```

THIS IS ALL COOL FOR MYPY

```
justin_bieber = Singer()
max_rock(justin_bieber, "Some crap")
```

MYPY SAYS:

```
abstract_example.py:34: error:
    Argument 1 to "max_rock"
      has incompatible type "Singer";
      expected "Rocker"`
```

ALTHOUGH SCALA HAS `abstract` (LIKE JAVA). THE PREFERRED WAY IS USING TRAITS

SCALA. ABSTRACT -> STAIRCASE. DUCHAMP

```scala
trait CanSing {
  def sing(tune: String): String = {
    tune
  }
}

case class Singer(name: String, band: String) extends CanSing

trait PlaysGuitar {
  def rock(tune: String): String
}

case class Rocker(name: String, band: String) extends CanSing with PlaysGuitar {
  override def sing(tune: String): String = {
    s"YEAH ${tune}"
  }

  def rock(tune: String): String = {
    s"🎸 ${tune} 🎸"
  }
}
```

```scala
object TraitsExamples {
  def maxRock(rocker: Rocker, tune: String): String =
    s"${rocker.rock(rocker.sing(tune))}"

  def guitarIt(player: PlaysGuitar, tune: String): String =
    s"${player.rock(tune)}"

  def main(args: Array[String]): Unit = {
    val geneSimmons = Rocker("Gene Simmons", "KISS")

    println(maxRock(geneSimmons, "I was made for loving you"))
    println(guitarIt(geneSimmons, "Strutter"))

    // val justinBieber = Singer("Justin Bieber", "himself")
    // maxRock(justinBieber, "some crap")
    //   traits.scala:24: error: type mismatch;
    //   found   : Singer
    //   required: Rocker
  }
}
```

IMPLICITS IN SCALA

THEY DON'T MAKE MUCH SENSE IN PYTHON[3] BUT IN SCALA THEY LET YOU DO A BIT OF MAGIC

[3] WE HAVE MONKEY-PATCHING, BUT IT BREAKS MYPY TYPE CHECKING THOUGH

```scala
object Something {
  private def doWhatever(config: Config) = {
    implicit val spark = SparkSession.builder
      .appName(Config.appName)
      .getOrCreate()

    Another().doCoolSparkyStuff(42)
  }
}

class Another {
  def doCoolSparkyStuff(num: Int)(implicit spark: SparkSession): Int = {
    ...
  }
}
```

AT FIRST LOOK THEY LOOK LIKE GLOBAL VARIABLES. BUT STRICT TYPE CHECKING AND SCOPING MAKE IMPLICITS TERRIBLY USEFUL

THIS SLIDE INTENTIONALLY LEFT BLANK BECAUSE TYPE CLASSES ARE TOO WEIRD

AFTER WORKING A BIT WITH SCALA
I HAVE DISCOVERED MANY, MANY
THINGS WE CAN DO IN PYTHON
EASIER AND BETTER

LEARN SOME SCALA AND COME
BACK TO PYTHON!

YOU CAN FIND CODE SAMPLES AND THE PRESENTATION SOURCE IN

GITHUB.COM/RBERENGUEL/SNAKES_AND_LADDERS

YOU CAN GET THE PRESENTATION FROM

SLIDESHARE.COM/RBERENGUEL/SNAKESANDLADDERS

References and sources

Scala growth graph

EPFL staircase by Miles Sabin

Snake picture

Snakes and ladders board

Twelfth Doctor gif: from Doctor Who

Pillars

Witch

Nude Descending a Staircase, No. 2 by Marcel Duchamp, Wikipedia

Fixes in production gif: from Buster Keaton's The General