# Conceptual Programming with Python

Thorsten Altenkirch

Functional Programming Laboratory
School of Computer Science
University of Nottingham

February 12, 2020

# History

**Since 2013** I am teaching G54PRG (now COMP4008): *Programming*

- Module for Master Students with no or very little programming experience.
- Previously used C# as programming language.
- I changed it to Python

**Since 2017** Teaching jointly with Isaac Triguero.

**Autumn 2019** Publish our book, based on jupyter python scripts.

**Since Autum 2019** A number of computerphile videos on youtube on topics covered in the book.

# Why Python?

- Python has a very simple syntax with very little overhead. It uses layout to represent structure which is very natural and easy to read.
- Python uses dynamic typing, this makes it easy to learn because you don't have to get your head around a static type system, but see below.
- Python allows you to use concepts from a variety of programming paradigms, including object oriented programming and functional programming.
- There are a number of tools which make Python easy to use, like *jupyter notebooks* which we are using.
- Python features a *toplevel* like many functional languages, which makes it easy to interactively explore the language.
- Python is very popular, which results in a number of libraries (APIs) available in Python, which often makes it the language of choice in practice.

# Why not Python?

- The fact that Python doesn't use static typing means that many errors which would be flagged by other languages go undetected and may cause hidden errors in the software. These also means that interfaces are not clearly defined making the development of large systems harder.
- Python makes it often hard to use modern concepts, like recursion, because you have to pay an unnecessary performance penalty.
- Python also lacks certain features, like a pattern matching and algebraic data types, making the representation often unnecessarily clumsy.
- The lack of types leads to certain design errors in Python, for example the decision to avoid characters and represent them as strings.

# Python's character madness

```
In : def swap(x) :
        return x[-1]+x[1:-1]+x[0]

In : swap("Python")

Out: 'nythoP'

In : swap([1,2,3])

TypeError:  unsupported operand type(s) for +:  'int' and 'list'
```

```
In : def swap (x) :
         return x[-1:]+x[1:-1]+x[:1]

In : swap("Python")

Out: 'nythoP'

In : swap([1,2,3])

Out: [3, 2, 1]
```

# Course structure

1. Python from the toplevel
2. Imperative programming
3. Recursion and backtracking
4. Object oriented programming
5. Pygame API

# Assessment

1. 4 courseworks on topics 1-4 (20 %)
   assessed (but not marked) in the lab.
2. Pygame group project (30 %)
   demo + peer assessment + prizes

3. Written exam (50%)
   what is the output of . . . ?

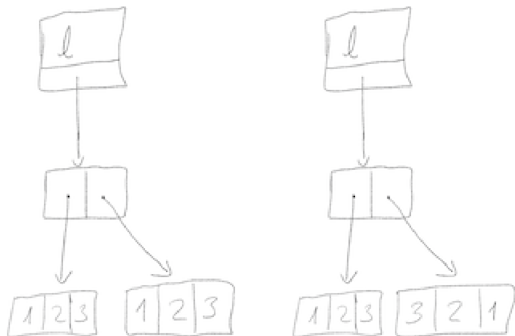
# Explaining sharing

```
In : l=[[1,2,3],[1,2,3]]

In : rotateRx(l[1])

In : l

  Out: [[1, 2, 3], [3, 1, 2]]
```



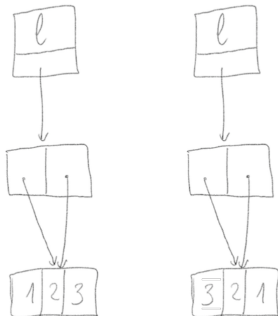
# How do I make this list?

```
In : l
Out: [[1, 2, 3], [1, 2, 3]]
In : rotateRx(l[1])
In : l
Out: [[3, 1, 2], [3, 1, 2]]
```

# Answer

```
In : lx = [1,2,3]

In : l = [lx,lx]
```



# The Halting problem in Python

Our assumption is that there is a Python function

```
def halts(fun,arg) :
  ...
```

which gets two string arguments: `fun` which contains a function definition and `arg` which is an argument and the function will return `true` if the Python program which we get when applying `fun` to the argument terminates and `false` otherwise.

# weird

```
def weird(fun) :
    def halts(fun,arg) :
        ...
    if halts(fun,fun) :
       while(true) :
           pass
       else :
           return
```

```
wstr =
"""
def weird(fun) :
...
"""
```

Now what happens if we run

```
weird(wstr)
```

# Towers of Hanoi

```
In : def hanoi(n,f,h,t):
         if n==0:
             return
         else:
             hanoi(n-1,f,t,h)
             print("Move disk from {} to {}".format(f,t))
             hanoi(n-1,h,f,t)


In : hanoi(4,"A","B","C")


Move disk from A to B
Move disk from A to C
Move disk from B to C
...
```

# Sudoku

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | | | 7 | | | | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | 2 | 8 | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

```
In : def solve() :
         global grid
         for y in range(0,9) :
             for x in range(0,9) :
                 if grid[y][x] == 0 :
                     for n in range(1,10) :
                         if possible(y,x,n) :
                             grid[y][x] = n
                             solve()
                             grid[y][x] = 0
                     return
         print_grid()
         input("More?")

In : solve()
```

# Exercise 4 : Boolean expressions

```
e1 = Or(Var("x"),Not(Var("x")))
e2 = Eq(Var("x"),Not(Not(Var("x"))))
e3 = Eq(Not(And(Var("x"),Var("y"))),
        Or(Not(Var("x")),Not(Var("y"))))
e4 = Eq(Not(And(Var("x"),Var("y"))),
        And(Not(Var("x")),Not(Var("y"))))
```

# Part 1 : Create objects

```
In : class Expr :
         pass

     class Not(Expr) :
         def __init__(self,e) :
             self.e = e
     class BinOp(Expr) :
         def __init__(self,l,r) :
             self.l = l
             self.r = r
     class And(BinOp) :
         pass
     class Or(BinOp) :
         pass
     class Eq(BinOp) :
         pass
     class Const(Expr) :
         def __init__(self,val) :
             self.val = val
     class Var(Expr) :
         def __init__(self,name) :
             self.name = name
```

# Part 2 : print expressions

```
In : print(e1)
x|!x

In : print(e2)
x==!!x

In : print(e3)
!(x&y)==!x|!y

In : print(e4)
!(x&y)==!x&!y
```

```
In : class Expr :

         def __str__(self) :
             return self.str_aux(0)

     class BinOp(Expr) :

         def __init__(self,l,r) :
             self.l = l
             self.r = r

         def str_aux(self,fix) :
             s=self.l.str_aux(self.fix)+\
               self.sym+self.r.str_aux(self.fix)
             if fix > self.fix :
                 return "("+s+")"
             else :
                 return s

     class And(BinOp) :

         fix = 2
         sym = "&"
```

# Tautology checker

**Exercise 04**: A proposition is called a tautology if it is always `true`. That is, the truthtable contains only `True` in the last column. Implement a method `isTauto` which determines whether the proposition is a tautology. E.g.

```
>>> e1.isTauto()
True
>>> e4.isTauto()
False
```

# vars and eval

```
In : class BinOp(Expr) :

         def vars(self) :
             return self.l.vars().union(self.r.vars())

         def eval(self,env) :
             return self.op(self.l.eval(env),self.r.eval(env))

     class And(BinOp) :

         fix = 2
         sym = "&"

         def op(self,l,r) :
             return l&r
```

# isTauto

```
In : class Expr :

         def isTauto(self) :
            tt = truthtable(list(self.vars()))
            for env in tt :
                if not self.eval(env) :
                    return False
            return True
```

# The sieve of Erathostenes

```
In : def nats(n) :
         yield n
         yield from nats(n+1)

In : def sieve(ns) :
         n = next(ns)
         yield n
         yield from (i for i in sieve(ns) if i%n != 0)

In : primes = sieve(nats(2))

In : show(primes)

2         More? y
3         More? y
5         More? y
7         More? y
11        More? y
13        More? y
```

# Observations

- You can teach programming concepts with Python.
- The pythonesque philosophy sometimes gets in your way but can be largely ignored.
- Dynamic typing is good for the beginner. They don't have to understand the static semantics, and if the program goes wrong its their fault!
- Do coursework demos in the lab, and give feedback in person.
- But don't give them their marks in the lab!
- Use the toplevel, and do not allow IDEs that don't integrate the toplevel.

# Observations

- Don't use slides! Live hacking is much more fun and can go wrong in interesting ways. Plus write on the ipad.
- Jupyter is good to record life hacking sessions for lecture notes.
- Important message: Copying code is evil. However, it is ok for a quick and dirty prototype but then refactor your code later.
- Implementing games with pygames is very popular and adds a fun component to the course.
- The only way to learn software engineering is by doing it. Games are a good subject for this.
- Written exams are useful. In particular you can check wether they can run python programs in their head.