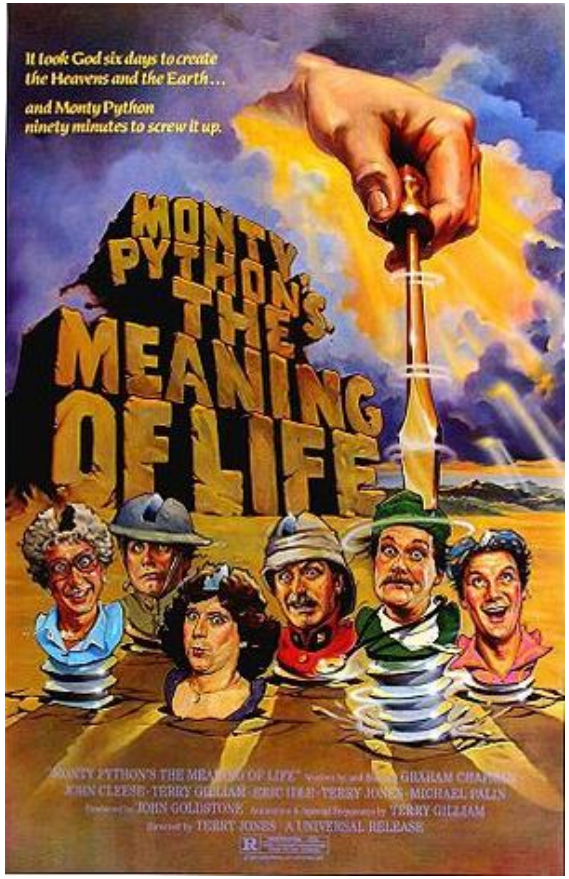
Functional Programming

- Functions are objects
- Lambda notation
- Functional programming
- Closure, map, filter, reduce



function definition

IMPERATIVE

```
function factI (n)
  local accumulator = 1
  for i = 1,n do
    accum = accumulator*i
  end
  return accum
end
```

Variable

trace of execution

factI(4) =
accumulator = 1
i = 1 accumulator = 1 * 1
i = 2 accumulator = 1 * 2
i = 3 accumulator = 2 * 3
i = 4 accumulator = 6 * 4
return 24

function definition

FUNCTIONAL

```
function factR (n)
  if n == 1 then
    return 1
  else
    return n*factR(n-1)
  end
end
```

No variables
Just param...
use recursion 4 loops

trace of execution

```
factR(4) =
4 * factR(3) =
      3 * factR(2) =
            2 * factR(1) =
                  1
            1
      2
6
24
```

Functions are first-class objects

Functions can be used as any other data type, eg:

- Arguments to function
- Return values of functions
- Assigned to variables
- Parts of tuples, lists, etc

```
>>> def square(x): return x*x
>>> def applier(q, x): return q(x)
>>> applier(square, 7)
49
```

Lambda Notation

Python's lambda creates anonymous functions

```
>>> lambda x: x + 1
<function <lambda> at 0x1004e6ed8>
>>> f = lambda x: x + 1
>>> f
<function <lambda> at 0x1004e6f50>
>>> f(100)
101
```

Lambda Notation

Be careful with the syntax

```
>>> f = lambda x,y: 2 * x + y
>>> f
<function <lambda> at 0x87d30>
>>> f(3, 4)
10
>>> v = lambda x: x*x(100)
>>> v
<function <lambda> at 0x87df0>
>>> v = (lambda x: x*x)(100)
>>> v
10000
```

Lambda Notation Limitations

- Note: only **one** expression in the lambda body; Its value is always returned
- The lambda expression must fit on one line!
- Lambda will probably be deprecated in future versions of python

  Guido is not a lambda fanboy

Functional programming

- Python supports functional programming idioms
- Built-ins for map, reduce, filter, closures, continuations, etc.
- These are often used with lambda

Example: composition

```
>>> def square(x):
        return x*x
>>> def twice(f):
        return lambda x: f(f(x))
>>> twice
<function twice at 0x87db0>
>>> quad = twice(square)
>>> quad
<function <lambda> at 0x87d30>
>>> quad(5)
625
```

Example: closure

```
>>> def counter(start=0, step=1):
        x = [start]
        def _inc():
            x[0] += step
            return x[0]
        return _inc
>>> c1 = counter()
>>> c2 = counter(100, -10)
>>> c1()
1
>>> c2()
90
```

map

```
>>> def add1(x): return x+1
>>> map(add1, [1,2,3,4])
[2, 3, 4, 5]
>>> map(lambda x: x+1, [1,2,3,4])
[2, 3, 4, 5]
>>> map(+, [1,2,3,4], [100,200,300,400])
map(+,[1,2,3,4],[100,200,300,400])
          ^
SyntaxError: invalid syntax
```

Example

```
>>> sentence = 'It is raining cats and dogs'
>>> words = sentence.split()
>>> print (words)
['It', 'is', 'raining', 'cats', 'and', 'dogs']

>>> lengths = map(lambda word: len(word), words)
>>> print (lengths)
[2, 2, 7, 4, 3, 4]
```

map

- + is an operator, not a function
- We can define a corresponding add function

```
>>> def add(x, y): return x+y
>>> map(add,[1,2,3,4],[100,200,300,400])
[101, 202, 303, 404]
```

- Or import the operator module

```
>>> from operator import *
>>> map(add, [1,2,3,4], [100,200,300,400])
[101, 202, 303, 404]
>>> map(sub, [1,2,3,4], [100,200,300,400])
[-99, -198, -297, -396]
```

filter function

- filter offers an elegant way to filter out all the elements of a list.
- In `filter(f,l)`
  - the 1st argument is a function f that returns a Boolean value, i.e. either True or False.
  - This function is applied to every element of the list *l*.
  - Only if f returns True will the element of the list be included in the result list.

filter function

```
>>> filter(odd, [1,2,3,4])
[1, 3]

>>> fib = [0,1,1,2,3,5,8,13,21,34,55]
>>> result = filter(lambda x: x % 2, fib)
>>> print (result)
[1, 1, 3, 5, 13, 21, 55]
>>> result = filter(lambda x: x % 2 == 0, fib)
>>> print result
[0, 2, 8, 34]
```

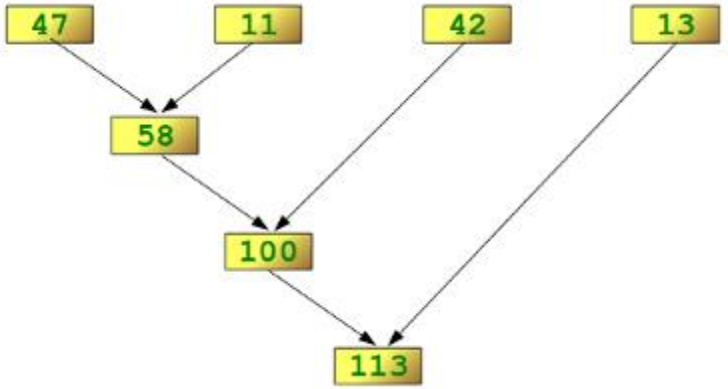
reduce function

- `reduce(func, seq)` continually applies the function `func()` to the sequence `seq`. It returns a single value.
- If seq = [s_1, s_2, s_3, ... , s_n], calling `reduce(func, seq)` works like this:
  - At first, the first two elements of `seq` will be applied to `func`, i.e. `func(s1,s2)`
  - The list on which `reduce()` works looks now like this: [func(s_1, s_2), s_3, ... , s_n]
- In the next step `func` will be applied on the previous result and the third element of the list, i.e. `func(func(s1, s2),s3)`
  - The list looks like this now: `[ func(func(s1, s2),s3), ... , sn ]`
- Continue like this until just one element is left and return this element as the result of `reduce()`

Illustration

```
>>> reduce(lambda x,y: x+y, [47,11,42,13])
113
```

- The following diagram shows the intermediate steps of the calculation:



reduce example

- Determining the maximum of a list of numerical values by using reduce:

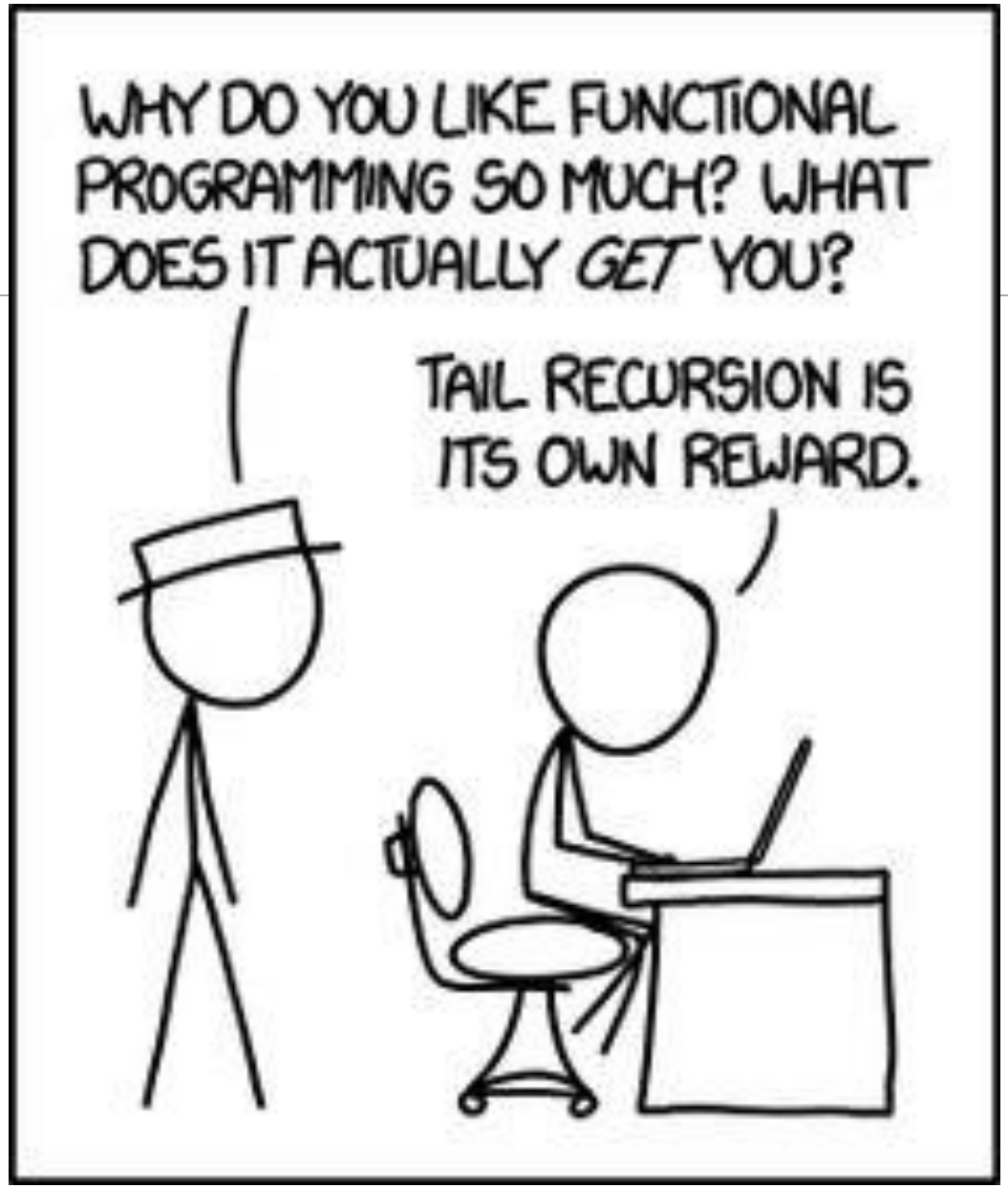
```
>>> f = lambda a,b: a if (a > b) else b
>>> reduce(f, [47,11,42,102,13])
102
>>>
```

- Calculating the sum of the numbers from 1 to 100:

```
>>> reduce(lambda x, y: x+y,range(1,101))
5050
```

filter, reduce

- The map, filter and reduce functions are also at risk of being dropped.


WHY DO YOU LIKE FUNCTIONAL PROGRAMMING SO MUCH? WHAT DOES IT ACTUALLY GET YOU?
TAIL RECURSION IS ITS OWN REWARD.