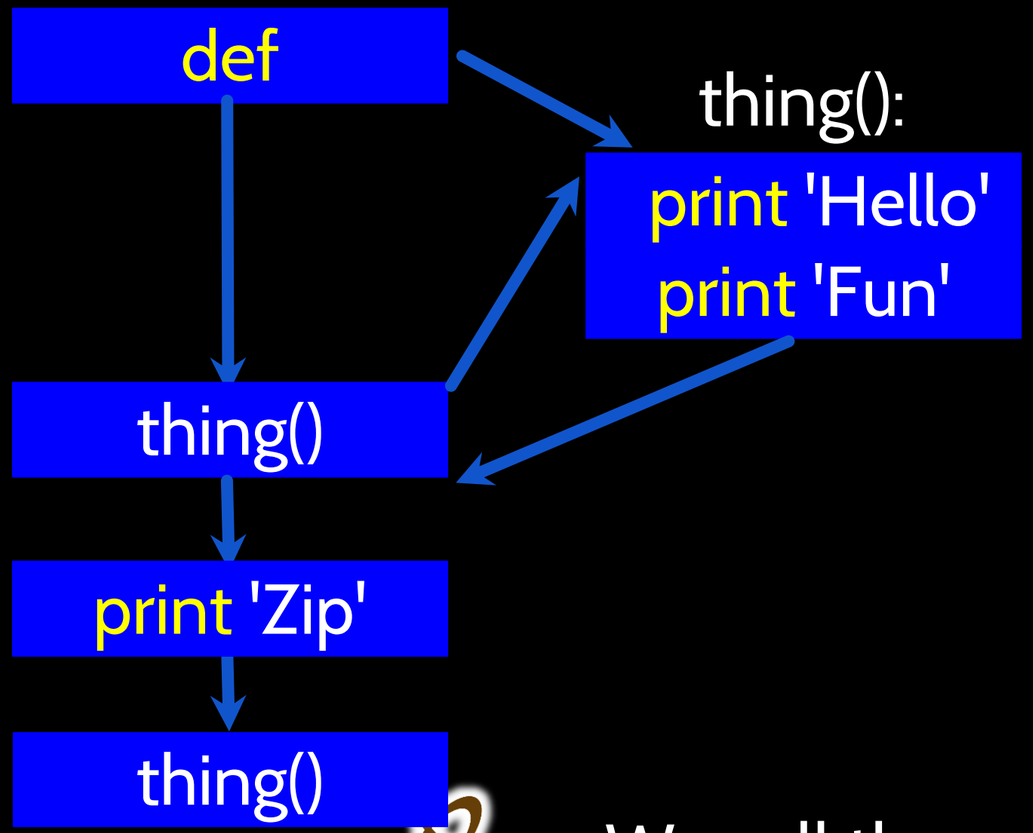
Learning Python

Session 4 - Functions

TechBoost
Boost Your Learning
www.techboost.in

Know BIG DATA
www.KnowBigData.com

Stored (and reused) Steps



Program:

```
def thing():
    print 'Hello'
    print 'Fun'

thing()
print 'Zip'
thing()
```

Output:

```
Hello
Fun
Zip
Hello
Fun
```

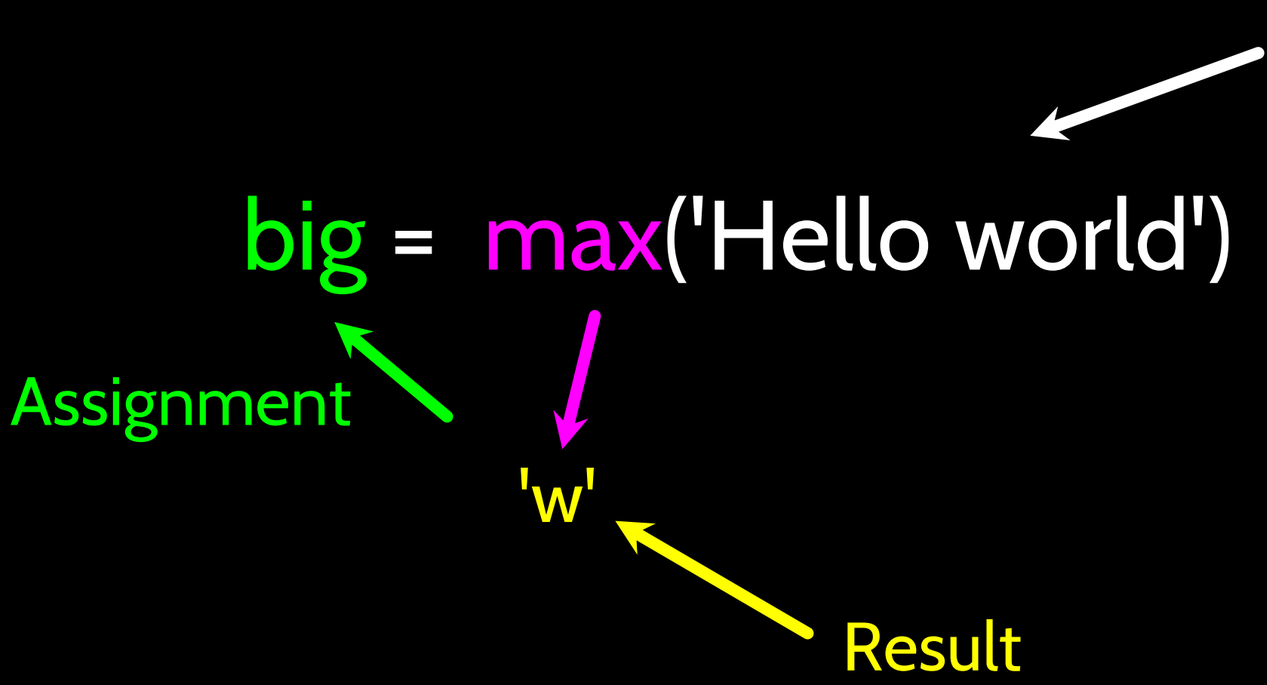
We call these reusable pieces of code “functions”

Python Functions

- There are two kinds of functions in Python.
  - Built-in functions that are provided as part of Python - raw_input(), type(), float(), int() ...
  - Functions that we define ourselves and then use
- We treat the built-in function names as "new" reserved words (i.e., we avoid them as variable names)

Function Definition

- In Python a function is some reusable code that takes arguments(s) as input, does some computation, and then returns a result or results
- We define a function using the def reserved word
- We call/invoke the function by using the function name, parentheses, and arguments in an expression



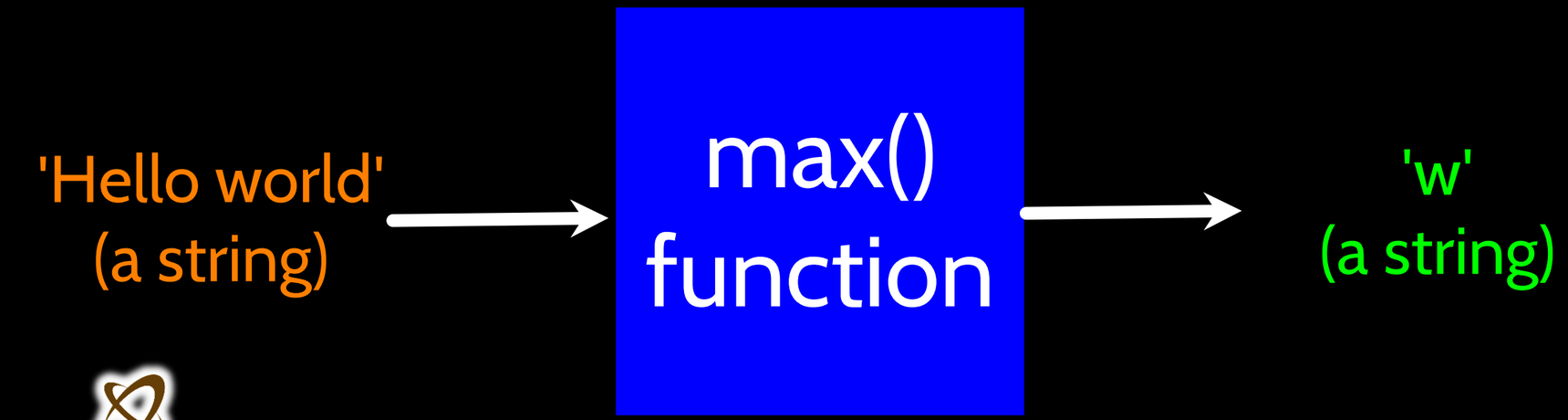

```
>>> big = max('Hello world')
>>> print big
w
>>> tiny = min('Hello world')
>>> print tiny

>>>
```

Max Function

```
>>> big = max('Hello world')
>>> print big
w
```

A function is some stored code that we use. A function takes some input and produces an output.

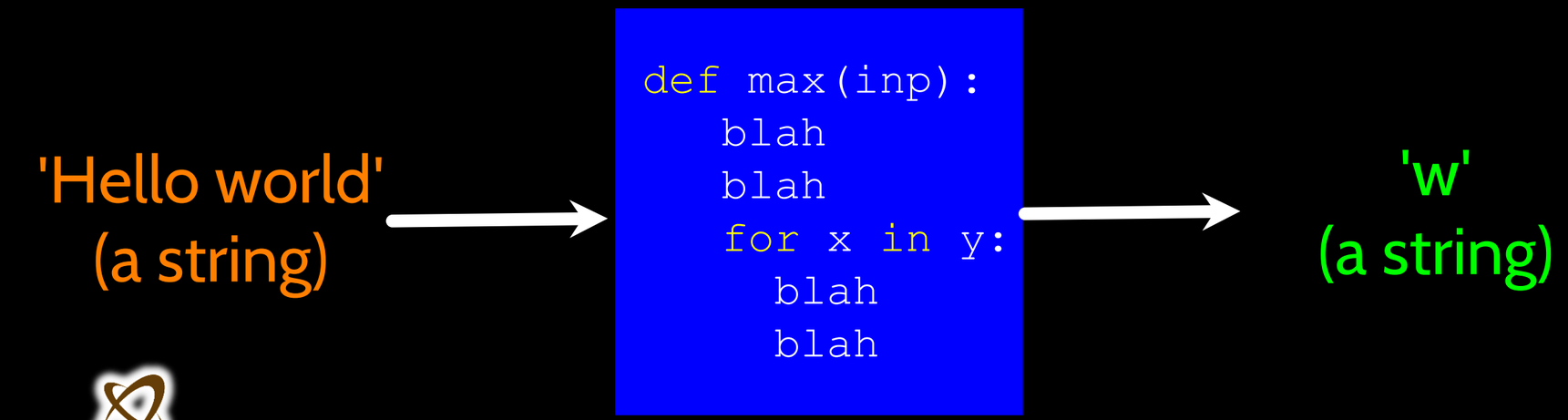


Guido wrote this code

Max Function

```
>>> big = max('Hello world')
>>> print big
w
```

A function is some stored code that we use. A function takes some input and produces an output.



Guido wrote this code

Type Conversions

- When you put an integer and floating point in an expression, the integer is implicitly converted to a float
- You can control this with the built-in functions int() and float()

```
>>> print float(99) / 100
0.99
>>> i = 42
>>> type(i)
<type 'int'>
>>> f = float(i)
>>> print f
42.0
>>> type(f)
<type 'float'>
>>> print 1 + 2 * float(3) / 4 - 5
-2.5
>>>
```

String Conversions

- You can also use int() and float() to convert between strings and integers
- You will get an error if the string does not contain numeric characters

```
>>> sval = '123'
>>> type(sval)
<type 'str'>
>>> print sval + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str'
and 'int'
>>> ival = int(sval)
>>> type(ival)
<type 'int'>
>>> print ival + 1
124
>>> nsv = 'hello bob'
>>> niv = int(nsv)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int()
```

Building our Own Functions

- We create a new function using the def keyword followed by optional parameters in parentheses
- We indent the body of the function
- This defines the function but *does not* execute the body of the function

```
def print_lyrics():
    print "I'm a lumberjack, and I'm okay."
    print 'I sleep all night and I work all day.'
```

print_lyrics():

```
print "I'm a lumberjack, and I'm okay."
print 'I sleep all night and I work all day.'
```

```
x = 5
print 'Hello'

def print_lyrics():
    print "I'm a lumberjack, and I'm okay."
    print 'I sleep all night and I work all day.'

print 'Yo'
x = x + 2
print x
```

```
Hello
Yo
7
```

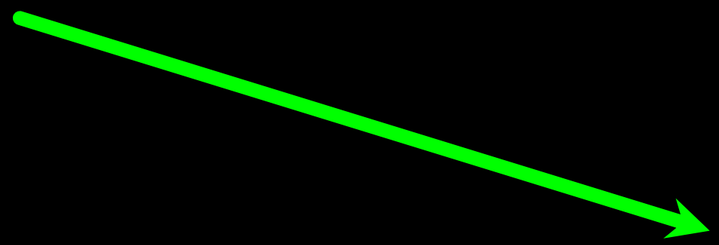
Definitions and Uses

- Once we have defined a function, we can call (or invoke) it as many times as we like
- This is the store and reuse pattern

```
x = 5
print 'Hello'

def print_lyrics():
    print "I'm a lumberjack, and I'm okay."
    print 'I sleep all night and I work all day.'

print 'Yo'
print_lyrics()
x = x + 2
print x
```



```
Hello
Yo
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
7
```

Arguments

- An argument is a value we pass into the function as its input when we call the function
- We use arguments so we can direct the function to do different kinds of work when we call it at different times
- We put the arguments in parentheses after the name of the function

```
big = max('Hello world')
```

Argument

Parameters

A parameter is a variable which we use in the function definition. It is a "handle" that allows the code in the function to access the arguments for a particular function invocation.

```
>>> def greet(lang):
...     if lang == 'es':
...         print 'Hola'
...     elif lang == 'fr':
...         print 'Bonjour'
...     else:
...         print 'Hello'
...
>>> greet('en')
Hello
>>> greet('es')
Hola
>>> greet('fr')
Bonjour
>>>
```

Return Values

Often a function will take its arguments, do some computation, and return a value to be used as the value of the function call in the calling expression. The return keyword is used for this.

```
def greet():
    return "Hello"

print greet(), "Glenn"
print greet(), "Sally"
```

```
Hello Glenn
Hello Sally
```

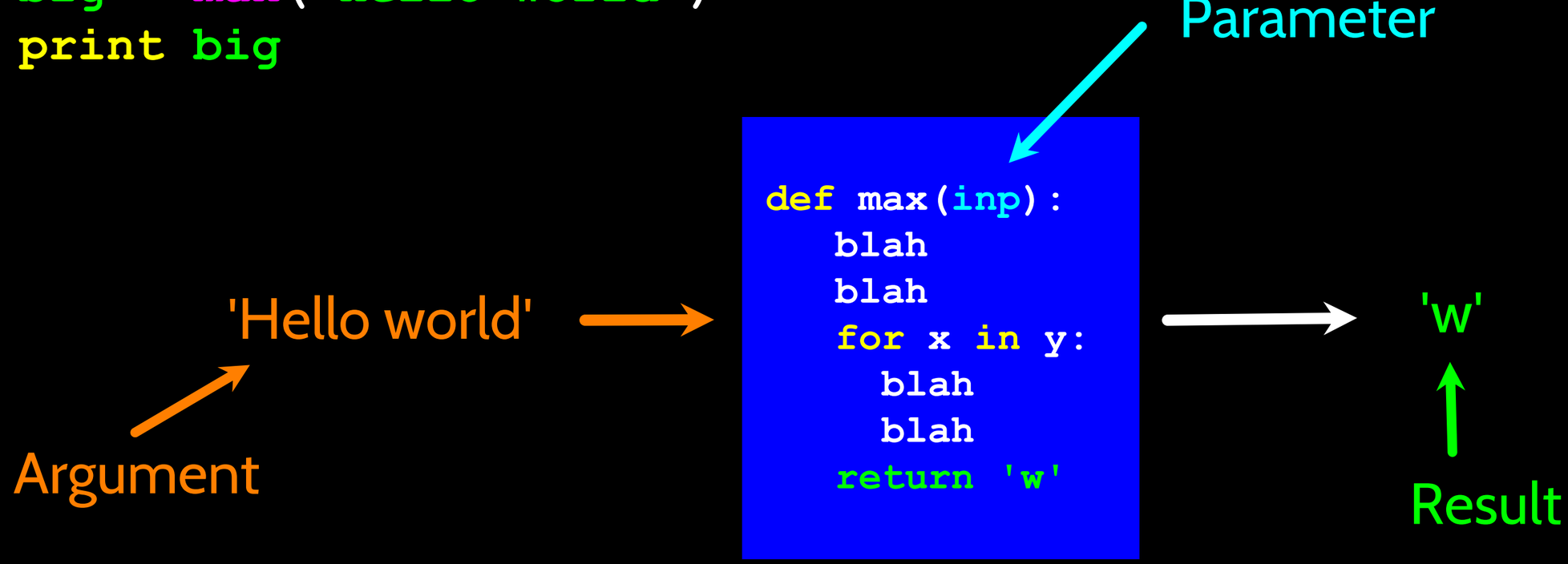
Return Value

- A “fruitful” function is one that produces a result (or return value)
- The return statement ends the function execution and “sends back” the result of the function

```
>>> def greet(lang):
...     if lang == 'es':
...         return 'Hola'
...     elif lang == 'fr':
...         return 'Bonjour'
...     else:
...         return 'Hello'
...
>>> print greet('en'),'Glenn'
Hello Glenn
>>> print greet('es'),'Sally'
Hola Sally
>>> print greet('fr'),'Michael'
Bonjour Michael
>>>
```

Arguments, Parameters, and Results

```
>>> big = max('Hello world')
>>> print big
w
```



Multiple Parameters / Arguments

- We can define more than one parameter in the function definition
- We simply add more arguments when we call the function
- We match the number and order of arguments and parameters

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print x
```

Multiple Return Values

```
def minMax(array):
    min = sys.maxint;
    max = -sys.maxint - 1;
    for i in array:
        if i < min:
            min = i;
        if i > max:
            max = i;
    return (min, max);
```

```
>>> minMax([1,2,3,4,5])
(1, 5)
```

Variable Arguments

```
def manyArgs(*arg):
      print "I was called with", len(arg), "arguments:", arg


>>> manyArgs(1)
I was called with 1 arguments: (1,)
>>> manyArgs(1, 2,3)
I was called with 3 arguments: (1, 2, 3)
```

Variable Keyworded Arguments

```
def greet_me(**kwargs):
    if kwargs is not None:
        for key, value in kwargs.iteritems():
            print "%s == %s" %(key,value)

>>> greet_me(name="yasoob", age="10")
name == yasoob

age == 10
```

Void (non-fruitful) Functions

- When a function does not return a value, we call it a "void" function
- Functions that return values are "fruitful" functions
- Void functions are "not fruitful"

Passing Function as argument

```
def circle(x):
    print x + " is circle";

def shape(shape, name):
    print "converting";
    shape(name);
```

```
>>> shape(circle, "sandeep");
converting
sandeep is circle
```

Passing Functions: Filter

- Executes a function on each element of array
- If the function returns True
- Puts it in output
- Distributable paradigm

```
def isEven(x):
    return x % 2 == 0;

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

Passing Functions: Map

- Executes a function on each element of array
- Returns the array containing output
- Distributable paradigm

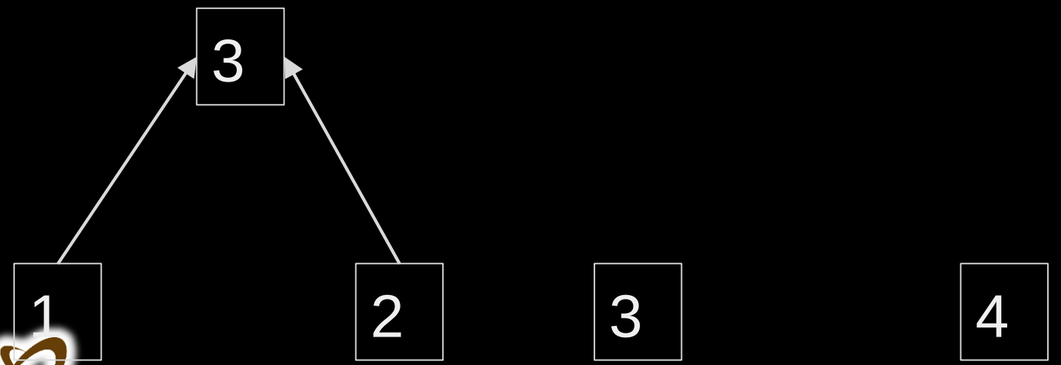
```
def my_map(x):
    return x * 2;
>>>arr = [1,2,3,4]
>>>map(my_map,arr );
[2, 4, 6, 8]
```

Passing Functions: Reduce

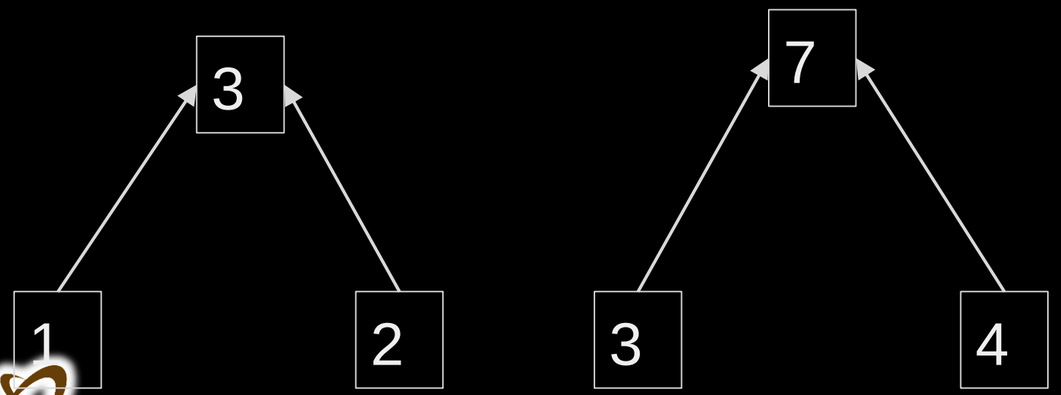
- Executes a function on two element of array
- Keeps executing recursively

```
def my_sum(x, y):
   return x +y;
>>>arr = [1,2,3,4]
>>>reduce(my_sum,arr );
10
```

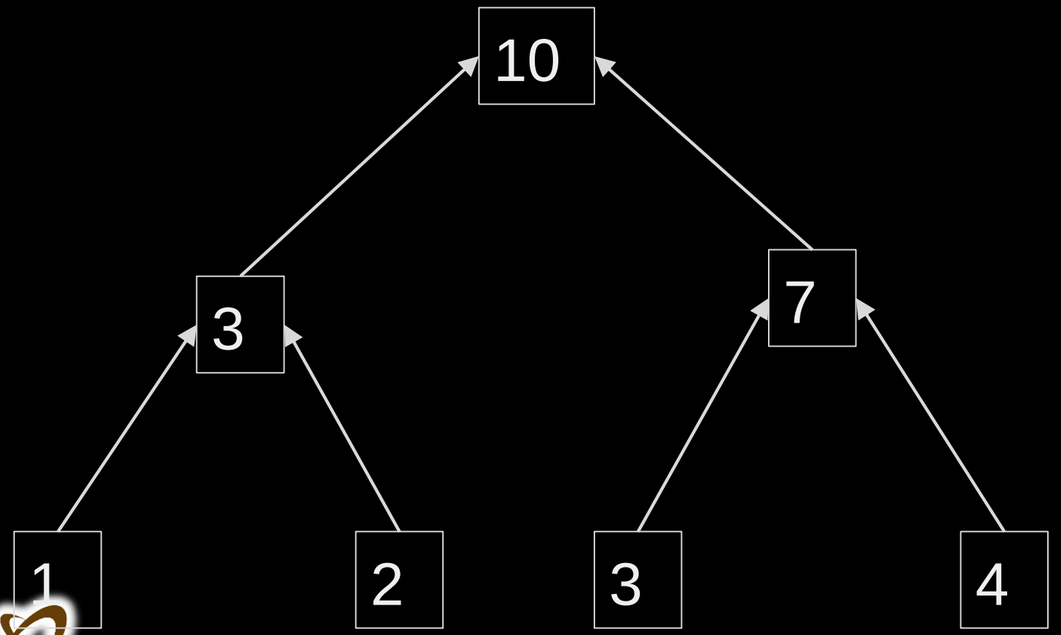
Passing Functions: Reduce



Passing Functions: Reduce



Passing Functions: Reduce



Match The results

1.Filter

2.Map

3.Reduce

Output has

A.Single Value

B.As many values as input

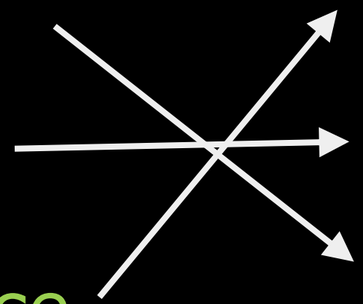
C.Less than or equal number of values as input

Match The results

1.Filter

2.Map

3.Reduce



Output has

A.Single Value

B.As many values as input

C.Less than or equal number of values as input

Lambda Function

1. Anonymous Function
2. Can be used quickly
3. Comes from functional programming

```
>>> def f (x): return x**2
>>> print f(8)
64
>>> g = lambda x: x**2
>>> print g(8)
64
```

Lambda Function: Map, Filter & Reduce

```
>>> foo = [2, 18, 9, 22, 17, 24, 8, 12, 27]
>>> print filter(lambda x: x % 3 == 0, foo)
[18, 9, 24, 12, 27]
>>> print map(lambda x: x * 2 + 10, foo)
[14, 46, 28, 54, 44, 58, 26, 34, 64]
>>> print reduce(lambda x, y: x + y, foo)
```

To function or not to function...

- Organize your code into “paragraphs” - capture a complete thought and “name it”
- Don’t repeat yourself - make it work once and then reuse it
- If something gets too long or complex, break it up into logical chunks and put those chunks in functions
- Make a library of common stuff that you do over and over - perhaps share this with your friends...

Exercise

Rewrite your pay computation with time-and-a-half for overtime and create a function called computepay which takes two parameters (hours and rate).

Enter Hours: 45
Enter Rate: 10
Pay: 475.0

475 = 40 * 10 + 5 * 15

Summary

- Functions
- Built-In Functions
  - Type conversion (int, float)
  - String conversions
- Parameters
- Arguments
- Results (fruitful functions)
- Void (non-fruitful) functions
- Why use functions?

Questions?