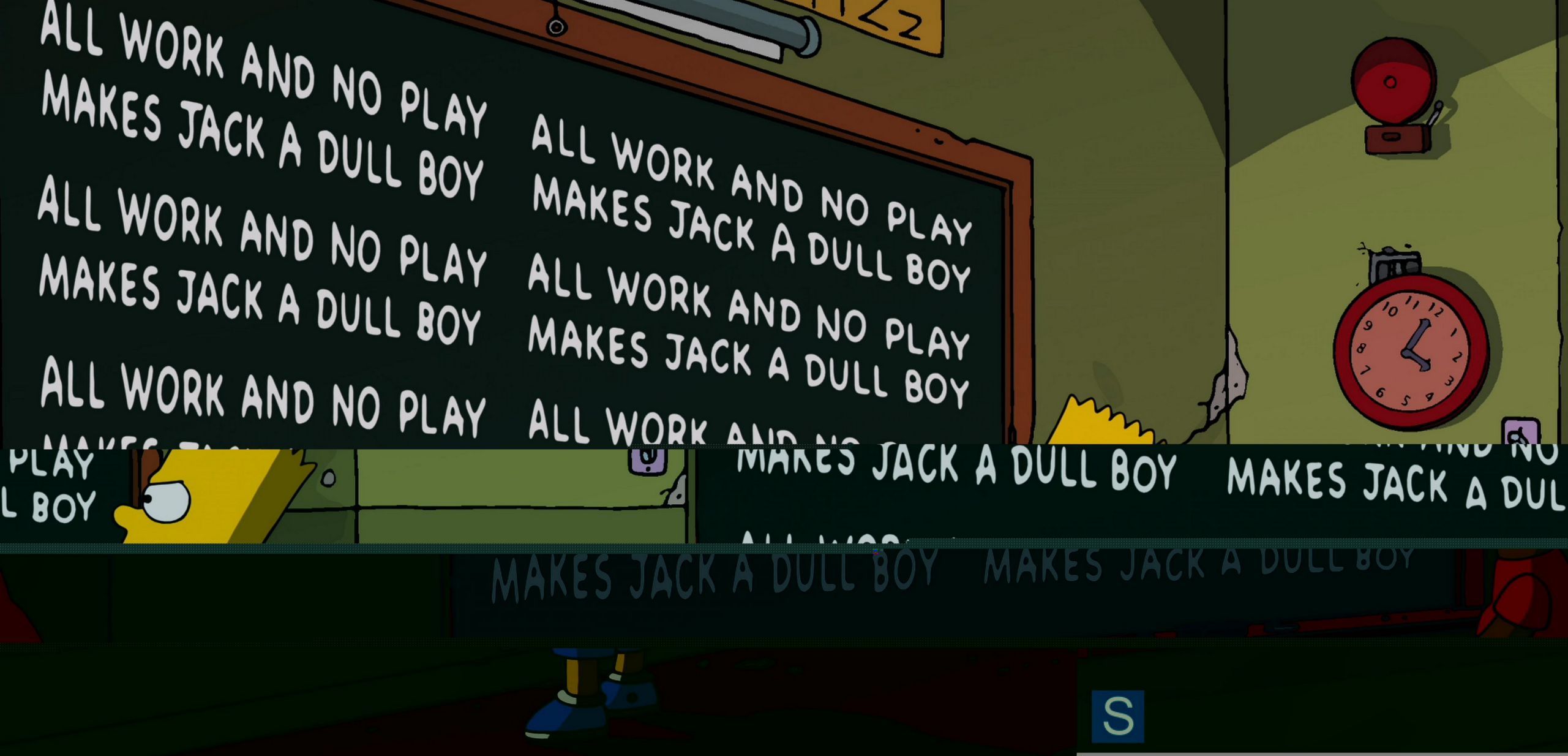




REVIEW

Computing machines are often used to automate repetitive tasks.



REVIEW: LOOPS

- A *loop* repeats a sequence of statements
- Two types
 - while loop
 - for loop



SENTINEL LOOPS

- A *sentinel loop* continues to process data until reaching a special value that signals the end.
- This special value is called the *sentinel*.

from ... import ...

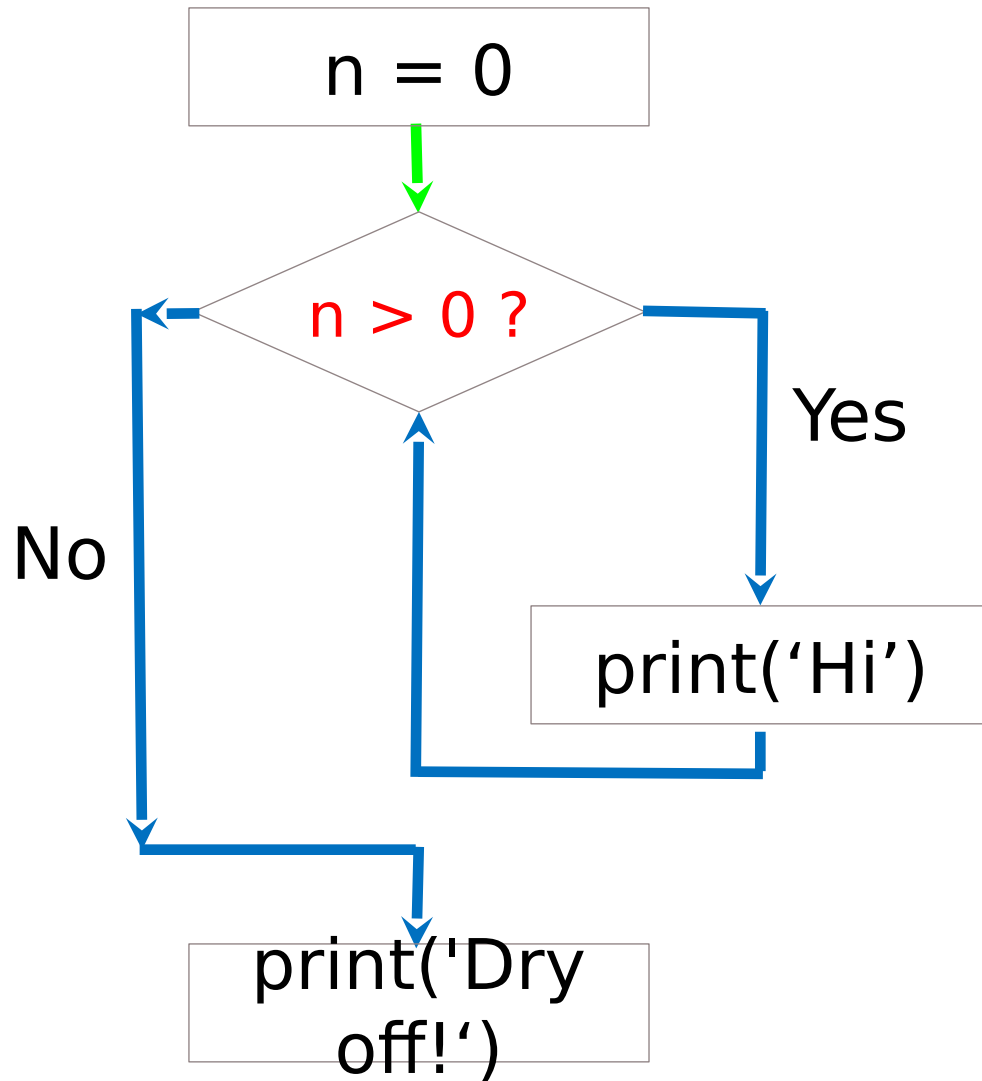
```
>>> from math import sqrt
```

```
>>> from math import *
```

```
>>> sqrt(25)
```

Difference? **Problem?**





What does this loop do?

```
n = 0
while (n > 0):
    print('Hi')
    print('Dry off!')
```



range

```
>>> range(1)
```

```
>>> range(25)
```

```
>>> range(2,10,2)
```



THE for LOOP

for *<variable>* **in** **range**(*<number of times>*):

<statement-1>

<statement-2>

...

<statement-n>

loop body



USING THE LOOP VARIABLE

- The *loop variable* picks up the next value in a *sequence* on each pass through the loop
- The expression **range(n)** generates a sequence of **ints** from **0** through **n - 1**

```
>>> for x in range(5):  
    print(x)
```



COUNTING FROM x THROUGH N

- The expression **range(low, high)** generates a sequence of **ints** from **low** through **high - 1**

```
for x in range(1, 6):  
    print(x)
```



ACCUMULATOR LOOP: SUMMATION

Compute and print the sum of the numbers between 1 and 5, inclusive

```
total = 0
```

```
for n in range(1, 6):
```

```
    total = total + n
```

```
print(total)
```

total += n



NESTED LOOPS

```
for <variable1> in range(<number of times>):  
    for <variable2> in range(<number of  
times>):  
        <statement-1>  
        <statement...>
```



NESTED LOOPS

```
while (<Condition>):  
    while (<Condition>):  
        <statement-1>  
        <statement...>
```



BREAK & CONTINUE

```
for letter in 'Python':
```

```
    if letter == 'h':
```

```
        break
```

```
    print(letter)
```

```
for letter in 'Python':
```

```
    if letter == 'h':
```

```
        continue
```

```
    print(letter)
```

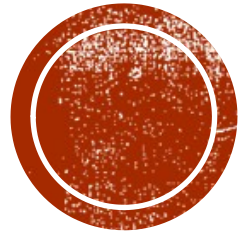


YET ANOTHER IMPORT STATEMENT

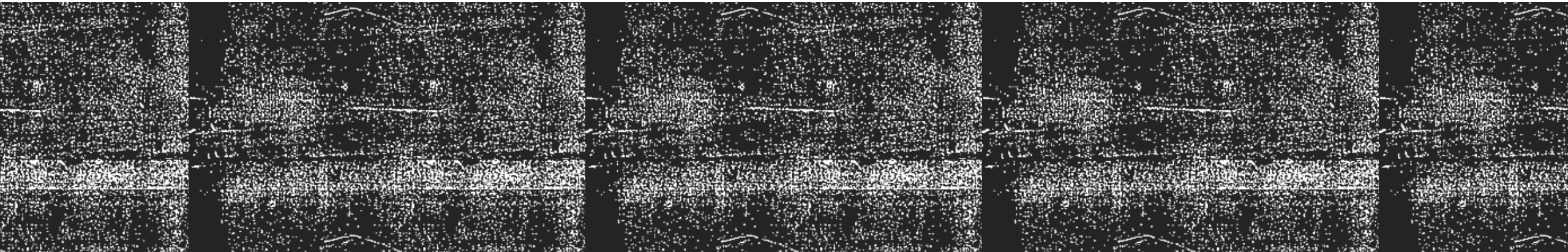
```
>>> import turtle as t
```

```
>>> bob = t.Turtle()
```





FRUITFUL FUNCTIONS



FUNCTIONS

The importance of functions:

- Break your code into separate, independent parts that will work together to solve the ultimate problem (**DECOMPOSITION**).
- Hide the details of your computation as long as you know what it produces (**ABSTRACTION**)



CONT...

The advantages of functions:

- Break your code into **simpler independent modules**
- These modules can be **reused** as many times as you like
- And they need to be **debugged only once**
- Keep your code **more organized and easier to**



FUNCTION PYTHON

- Defining functions:

Keyword

```
def function_name(arguments):  
    function_body  
    ...
```

0 or more
parameters/arguments

Calling functions:

```
function_name(arguments)
```



VOID FUNCTIONS

In several programming languages derived from C and Algol68, is the type for the result of a function that returns normally, but does not provide a result value to its caller.



VOID FUNCTIONS

```
def greeting(name):  
    print('hi ',name)
```

```
greeting('Haven')  
greeting('Robel')
```

```
def maximum(x,y):  
    if(x > y):  
        print(x)  
    else:  
        print(y)  
maximum(2,4)
```



<type 'NoneType'>

- Void functions might display something on the screen or have some other effect, but they don't have a return value.
- If you try to assign the result to a variable, you get a special value



FRUITFUL FUNCTIONS

A function that **returns** a value to the caller



RETURN VS PRINT

```
def greeting(name):  
    print('hi ',name)
```

```
greeting('Haven')  
greeting('Robel')
```

```
def maximum(x,y):  
    if(x > y):  
        return x  
    else:  
        return y  
maximum(2,4)
```



DEAD CODE

When a return statement executes, the function terminates without executing any subsequent statements.

Code that appears after a return statement, or any other place the flow of execution can never reach, is called **dead code**.



SCOPE

A variable can have something called a '*scope*'. This refers to its accessibility.



SCOPE

A variable can be declared in one of two ways...

When it is accessible from all
parts of a program
(ie. Anywhere!)

A **GLOBAL** Variable.

When it is accessible **ONLY**
within a function/procedure.

A **LOCAL** Variable.



EXAMPLE PROGRAMS

```
name = 'Mary'  
print(name)
```

```
def Greeting():  
    name = 'Mary Poppins'  
    print(name)
```

```
print(name)
```



GLOBAL VARIABLES

```
name = 'Mary Poppins'
```

```
def Greeting():  
    print(name)
```

```
Greeting()
```



GLOBAL VS LOCAL

```
num = 6
```

```
def multi():  
    num = 6  
    num = num * 3  
    print('local variable: ', num)
```

```
multi()  
print('Global: ', num)
```

Local: ?

18

Global: ?

6



BEST PROGRAMMING PRACTICES

Avoid use of global variables in favour of `local` variables.



OPTIONAL(DEFAULT) PARAMETERS

```
def print_name(fname, lname, reverse):  
    if(reverse):  
        print(lname, ' ', fname)  
    else:  
        print(fname, lname)
```



FUNCTION CALLING: POSITIONAL

The most common method, which is the only one we have used thus far, is called positional—the first parameter is bound to the first actual argument, the second to the second



FUNCTION CALLING: KEYWORD ARGUMENTS

Python also supports keyword arguments, in which formals are bound to actuals using the name of the formal parameter



BOOLEAN FUNCTIONS

- Functions that return Boolean values
- Convenient for hiding complicated tests



BOOLEAN FUNCTIONS

```
def is_divisible_by_2(num):  
    if(num % 2 == 0)  
        return True  
    else:  
        return False
```



BOOLEAN FUNCTIONS

```
def is_divisible_by_2(num):  
    if(num % 2 == 0)  
        return True  
    return False
```



DEBUGGING IN FUNCTIONS

- Preconditions
- Postconditions



COMMON PROBLEMS

- Indentation issues
- Parameter naming and passing

